Towards the Impact of Design Flaws on the Resources used by an Application

Cristina Marinescu, HPC Center, West University of Timişoara; "Politehnica University" Timişoara
Şerban Stoenescu, "Politehnica University" Timişoara
Teodor-Florin Fortiş, HPC Center, West University of Timişoara

One major research direction in cloud computing deals with the reduction of energy consumption. This can be seen as an optimization problem that must be addressed both at the hardware and the software (*i.e.*, application) level. At the software level, optimizing energy consumption is usually related with scaling down the resources (*e.g.*, memory, CPU usage) required for running an application. In this context, we can make the assumption that the presence of design flaws in the implementation of a software system may lead to a suboptimal resource usage. Our investigations on the impact of several design flaws on the amount of resources used by an application indicate that the presence of design flaws has an influence on memory consumption and CPU time, and that proper refactoring can have a beneficial influence on resource usage.

Categories and Subject Descriptors: Computer systems organization [**Architectures**]: Distributed architectures—*Cloud computing*; Software and its engineering [**Software creation and management**]: Software development process management—*Software development methods*

Additional Key Words and Phrases: Design flaws, Resource usage, Energy optimization, Energy consumption, Cloud computing

1. INTRODUCTION

Over the last years, cloud computing has become an increasingly widespread approach for deploying software systems. Consequently, an ever growing number of applications have been deployed on cloud infrastructures. However, hiring computing power for running an application in a cloud raises different issues related with energy consumption [Vouk 2008]. In this context, an important research direction in cloud computing deals with the issue of minimizing energy consumption, by optimizing resource usage [Lee and Zomaya 2012].

Energy consumption may be reduced by optimizing the hardware side (*e.g.*, by using more energy efficient cooling systems). However, in many cases the optimization can be even more cost-effective if, additionally, it addresses the software side, too. This can be done by reducing the used resources (*e.g.*, memory, CPU time) claimed by the applications which run in the cloud since it seems that energy consumption is affected by the used resources [Fan et al. 2007a].

The optimal usage of computing resources is an important factor of external quality. Therefore, in order to assess and control it, this quality factor must be put in relation with the internal quality criteria [ISO/IEC 2001]. One approach for quantifying the internal quality of an application is the Factor-Strategy Model [Marinescu and Raţiu 2004]. This model captures deviations from design rules and principles of software design in terms of design flaws (see [Fowler et al. 1999; Riel 1996]). Over the last years various empirical studies have shown that code entities affected by design flaws are

This work was partially supported by the grant of the European Commission FP7-REGPOT-CT-2011-284595 (HOST). The views expressed in this paper do not necessarily reflect those of the corresponding project consortium members.
Author's addresses: C. Marinescu and T.-F. Fortiş, HPC Center, West University of Timişoara, bvd. V. Parvan 4, 300223 Timişoara, Romania; email: cristinam@hpc.uvt.ro, fortis@info.uvt.ro; C. Marinescu and Şerban Stoenescu , "Politehnica" University Timişoara, Department of Computer Science, bvd. V. Parvan 2, 300223 Timişoara, Romania.

harder to maintain [Deligiannis et al. 2003], change more often [Khomh et al. 2009] and exhibit more defects [Li and Shatnawi 2007; Olbrich et al. 2010] than other entities which do not reveal design flaws. Therefore, a legitimate hypothesis is that one of the factors that may influence the amount of computing resources used by an application is the quality of its design.

In this paper we start an investigation for the following research question: does the existence of design flaws in the implementation of a software system lead to a suboptimal resource usage? This research question is particularly relevant considering the increasing number of cloud providers that offer solutions for deploying object-oriented applications to the cloud. Additionally, there is potential interest in the HPC area, as energy efficiency is a must for achieving *"petaFLOPS computational speeds and beyond"* [Song et al. 2009].

To the best of our knowledge there are no previous results that investigate if the presence of design flaws is correlated with an increased usage of computing resources.

The paper is structured as follows: in Section 2 we relate our investigation to previous work. In the first part of Section 3 we present the addressed research question, as well as the impact of a well-known design flaw towards the resource usages of a very simple program. We continue with an investigation towards the impact of design flaws against a well know open source software system. We end the section by pointing out the results of the study. The threats to the validity are presented in Section 4. In the last section (Section 5) we summarise the results and hint towards future work.

2. RELATED WORK

In this section we relate our work to the two domains we are investigating: energy optimization and design flaws.

2.1 Energy optimization

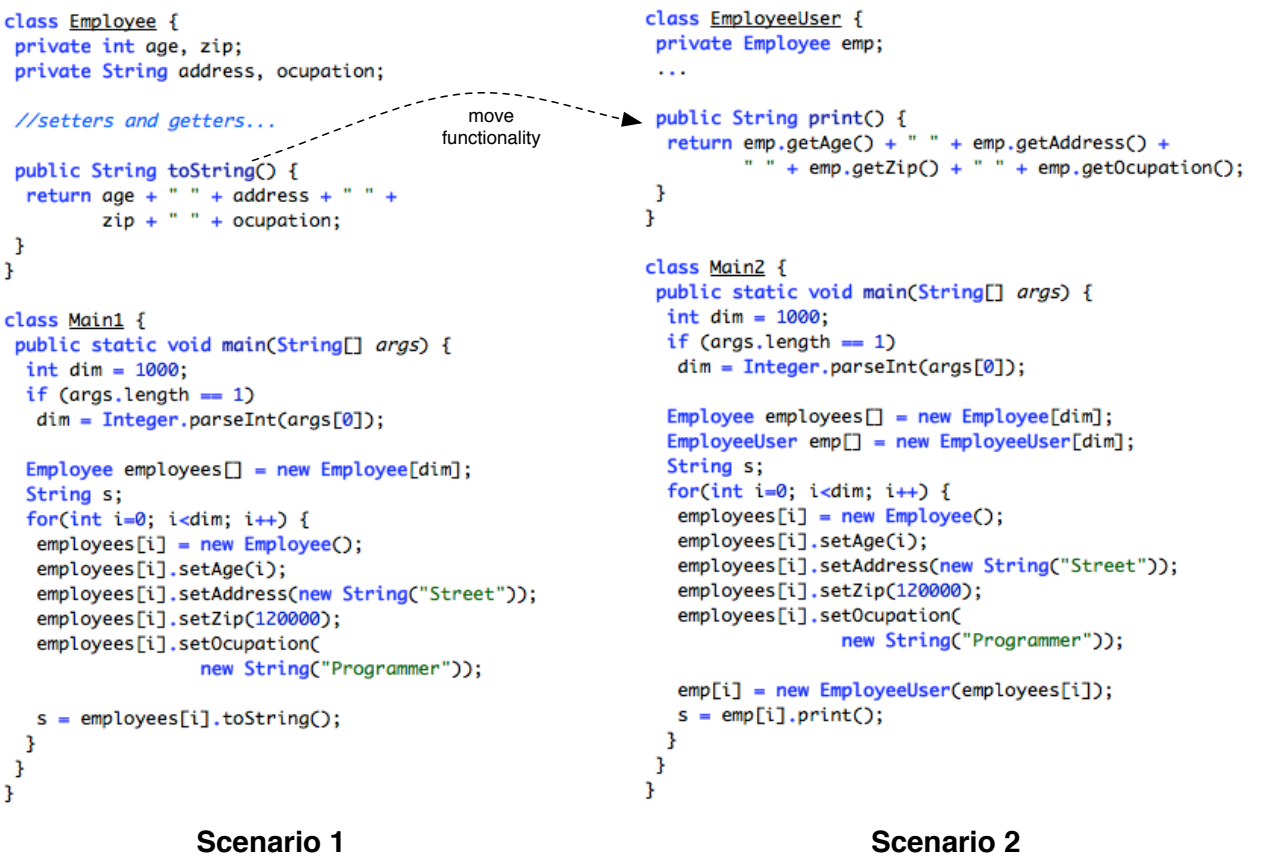
Energy optimization at the application-level is a topic that has lately gained much attention from both the research community, as well as the industry. According to [Fan et al. 2007a] energy consumption scales linearly with resource utilization. Therefore, we can consider the memory and CPU time used by an application as valuable metrics in the context of energy reduction.

Hindle [Hindle 2012] has performed a study that investigates the correlation between software changes and power consumption. His results show performance optimization across the versions of the investigated systems. The major difference to the work presented in our paper is that we manually refactored the available source code in order to reduce the number of the exhibited design flaws. By contrast, in the work of Hindle various measurements were performed on different versions of the analyzed systems without isolating changes made for system improvement.

Grosskop and Visser [Grosskop and Visser 2013] proposed an energy model of an application and suggested some optimizations at the application level. However, none of the proposed optimization are related to the presence of design flaws in the source code of the application.

2.2 Design flaws

Design flaws are deviations from the principles, patterns and best practices of software design, like the ones presented in [Riel 1996; Gamma et al. 1995; Martin 2008]. Fowler defined in [Fowler et al. 1999] a set of 22 design flaws which are considered to hamper the maintenance of object-oriented software systems. Most of the time it is desirable to get rid off the flawed entities and in order to do this the first step is to find the existing flawed entities within the systems. Finding manually flawed entities is time-consuming and this was the main reason automatic detection techniques of design flaws appeared. Probably the most used automatic approach for finding entities affected by various design flaws is the metrics-based technique. Currently there are many design flaws that can be detected automatically




```
class Employee {
 private int age, zip;
 private String address, ocupation;

 //setters and getters...

 public String toString() {
  return age + " " + address + " " +
         zip + " " + ocupation;
 }
}

class Main1 {
 public static void main(String[] args) {
  int dim = 1000;
  if (args.length == 1)
   dim = Integer.parseInt(args[0]);

  Employee employees[] = new Employee[dim];
  String s;
  for(int i=0; i<dim; i++) {
   employees[i] = new Employee();
   employees[i].setAge(i);
   employees[i].setAddress(new String("Street"));
   employees[i].setZip(120000);
   employees[i].setOcupation(
                  new String("Programmer"));

   s = employees[i].toString();
  }
 }
}
```

```
class EmployeeUser {
 private Employee emp;
 ...

 public String print() {
  return emp.getAge() + " " + emp.getAddress() +
         " " + emp.getZip() + " " + emp.getOcupation();
 }
}

class Main2 {
 public static void main(String[] args) {
  int dim = 1000;
  if (args.length == 1)
   dim = Integer.parseInt(args[0]);

  Employee employees[] = new Employee[dim];
  EmployeeUser emp[] = new EmployeeUser[dim];
  String s;
  for(int i=0; i<dim; i++) {
   employees[i] = new Employee();
   employees[i].setAge(i);
   employees[i].setAddress(new String("Street"));
   employees[i].setZip(120000);
   employees[i].setOcupation(
                  new String("Programmer"));

   emp[i] = new EmployeeUser(employees[i]);
   s = emp[i].print();
  }
 }
}
```

Fig. 1: The two experimentation scenarios.

(see [Khomh et al. 2009], [Li and Shatnawi 2007]) and different tools like *inFusion* [1] accompany the extraction of design flaws. Usually the tools parse the source code in order to extract the necessary information.

There are many recent empirical investigations, like the ones from [Deligiannis et al. 2003], [Khomh et al. 2009] and [Li and Shatnawi 2007], which show that design flaws have a strong negative influence on external quality factors like number of changes and/or defects. However, as already mentioned, we are not aware of any studies that investigate the impact of design flaws on the computational resources (memory, CPU) used by an application.

2.3 Cloud computing energy efficiency

In the context of cloud computing, energy efficiency offers important research challenges and issues [Vouk 2008; Zhang et al. 2010]. Different approaches for a "*Green Cloud*" exist, like the Green Cloud simulator[2] [Kliazovich et al. 2010], supported by the University of Luxemburg, or the Green Cloud

[1] http://www.intooitus.com/products/infusion
[2] http://greencloud.gforge.uni.lu/

Case Study	File(.class)	Size(bytes)
No Data Class	Employee	1096
	Main	824
Data Class	Employee	833
	EmployeeUser	667
	Main	953

Table I. : Dimensions of the involved `.class` files.

Project[3] [Garg et al. 2011], from the CLOUDS Laboratory of the University of Melbourne. An overview of power and energy management in data centers and cloud computing was offered by Beloglazov et al [Beloglazov et al. 2010], showing that *"Cloud computing naturally leads to power efficiency"* by providing a series of energy-oriented characteristics, including scaling up and down of resources, an approach that was considered in the context of our research.

Different researches directions also exists, including Virtual Machine (VM) placement and selection, together with appropriate selection policies [Garg et al. 2011; Beloglazov et al. 2012; Kliazovich et al. 2013] by employing power-aware scheduling mechanisms, provisioning and management of resources.

3. CASE-STUDY SETUP

The different approaches for an energy-aware cloud computing that currently exist are rather investigating optimization of cloud resource usage, by exploiting the energy-oriented characteristics of cloud computing [Beloglazov et al. 2010]. Even if there is a clear relationship between power consumption and CPU utilization [Fan et al. 2007b], there are few investigations on the impact of design flaws on the computational resources (*e.g.*, memory, CPU time) used by an application.

In order to address this intriguing research direction we arranged and performed two experimental studies, which are described in this Section.

3.1 A small case study: the Data Class design flaw

The goal of this study is the measurement of the influence of the *Data Class* [Riel 1996] design flaw on the used resources. *Data Classes* are mainly data containers which expose data instead of providing significant functionality. We choose to start our investigation with the influence of the *Data Class* flaw as previous studies have shown that this flaw has a very large lifespan in software projects [Peters and Zaidman 2012].

As shown in Figure 1, our experiment starts with an `Employee` class which contains four data members, as well as the full set of accessor methods (getters and setters) for the data members. In the first scenario (see left side of Figure 1), we also defined a simple service for the `Employee` class that returns its string representation, by concatenating the representation of its four data members.

In the second scenario, we created the `EmployeeUser` class which just aggregates an `Employee` object. However, the key element of the second scenario is the transformation of `Employee` in a pure *Data Class* by moving its sole service to the `EmployeeUser` class. Table I summarizes the size of the generated `.class` files, for all the classes involved in the two experimentation scenarios. The results indicate that the storage space is significantly larger in the second scenario.

In order to run the experiment for the two scenarios we defined for each a *driver class* (*i.e.*, `Main1` and `Main2`). Each of the two *drivers* is doing a very simple job: creates a large number of objects and calls the service. Thus, in *Scenario 1* we create `Employee` objects and call the `toString()` method,

[3]http://www.cloudbus.org/greencloud/

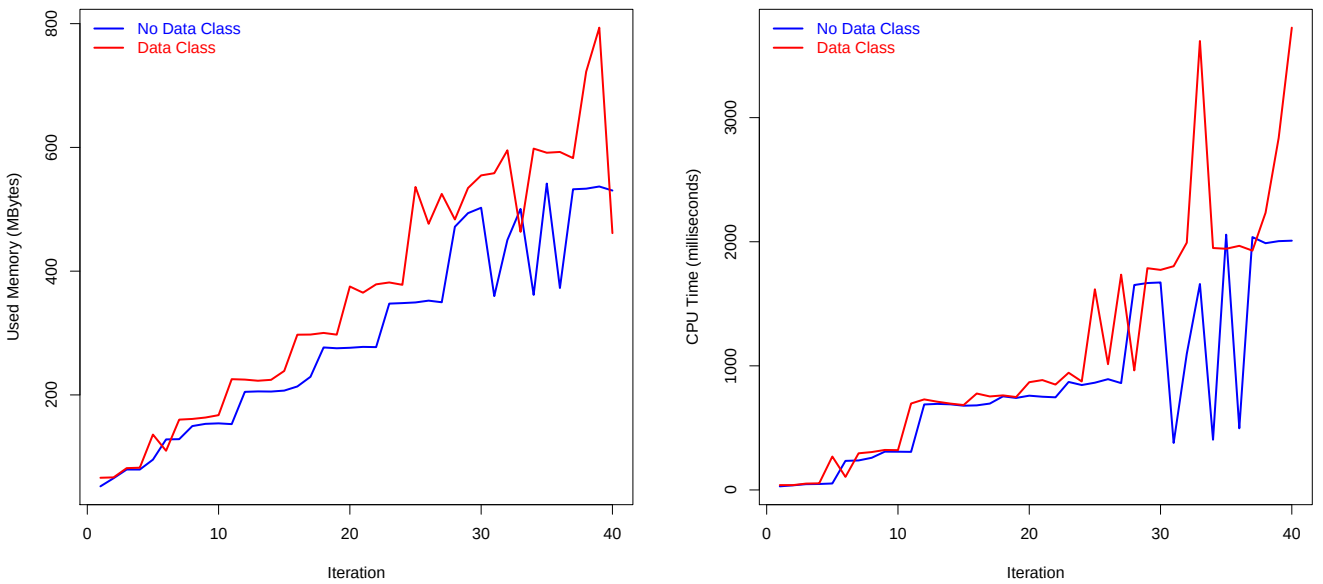


Fig. 2: Resource usages for the first experiment.

while in *Scenario 2* we create `EmployeeUser` objects, and call the `print()` service. For each scenario, we executed 40 iterations on a MacBook Pro with an Intel Core i5 2.53 GHz processor and 4GB of RAM. In the first iteration we instantiate 100 000 objects, and in the following iteration we increase the number of instantiated objects by another 100 000.

At each iteration we measured the *resident memory* and the *CPU time* used by the two versions of the target program, by using the *Hyperic Sigar*[4] tool. The results are summarized in Figure 2, showing that, in most cases that are running in *Scenario 2* (*i.e.*, the one that contains a *Data Class*) the usage of *memory* and *CPU time* is larger.

Next, based on the obtained data, we will answer the following research question: *Do the runs of Scenario 1 tend to use fewer resources than the runs of Scenario 2?* We consider that a run tends to exhibit a particular property if the chances of fulfilling that property are greater than 50%. We answer this research question by employing the proportion test and running it in R [R Development Core Team 2010].

According to the data from Figure 2, for 37 of the 40 runs the used memory of the program from *Scenario 1* is less than the used memory of the program from *Scenario 2* and for 36 of the 40 runs the CPU Time of the program from *Scenario 1* is smaller than the CPU Time of the program from *Scenario 2*. We firstly employed the following statistical test: *prop.test(37, 40, 0.5, alternative="greater")*, where the first parameter denotes the number of the runs of *Scenario 1* where the used memory was lower than the corresponding run of *Scenario 2*, the second parameter denotes the total number of runs and

[4]http://www.hyperic.com/products/sigar

0.5 denotes the true probability for a run of *Scenario 1* to use less memory than a run of *Scenario 2*. Regarding the CPU Time, we run *prop.test(36, 40, 0.5, alternative="greater")* because in 36 of the cases the CPU Time of the program from *Scenario 1* is lower than the CPU Time of the program from *Scenario 2*. Since in both of the cases p-value is lower than 0.05 (9.055e-08 and respectively 4.755e-07) we can conclude that running *Scenario 1* tends to use less resources than *Scenario 2*.

Since we decided to instantiate a very large number of objects, at first sight it may seem that the situation described with this first experiment setup is rather rare. In this context we want to emphasize that the investigated situation may occur quite frequently in the case of an application which was deployed in the cloud, as the number of objects gets multiplied once the application is scaling up due to a larger number of users that simultaneously access it. For example, instead of instantiating for each user the first version of the `Employee` class, we allow the instantiation of the second version of the *Employee* class (*i.e.*, the one exhibiting the *Data Class* design flaw) as well as the `EmployeeUser` class, then this situation will by multiplied by the number of the users that simultaneously access the application.

3.2 JHotDraw

In the first experiment we used a rather naive example merely to investigate the impact of having a large number of instances of classes where data and functionality are separated, versus the more desirable case of working with instances of a single class that encapsulates data and provides a service based on those data. However, we are aware that an increase of the needed resources may also be encountered in the case of other design flaws. Therefore, we designed and executed a second experiment in order to investigate the relation between entities affected by design flaws and resource usages against a *real application*.

For this second experiment we used *JHotDraw* 5.4 as a subject system. *JHotDraw* is a Java GUI framework for technical and structured graphics. Our choice for this system was based on the fact that it is the subject of various empirical analyses like the one found in [Bavota et al. 2013].

3.2.1 *The experiment.* In order to setup the experiment, we needed a system with a reasonable history in order to increase the chances to spot design flaws, as it is known that the number of the design flaws is growing together as the system gets older [Peters and Zaidman 2012]. Additionally, we decided to chose a version that is the last before a major change as this increases the chances of working on a stable release.

The experiment was performed as follows:

—we executed all the available tests for the *JHotDraw* system, while measuring memory consumption and CPU time using the *Hyperic Sigar* tool.

—we used *inFusion* 1.6 to detect design problems in *JHotDraw*.

—based on *inFusion*'s findings we performed an extensive manual refactoring process, with the goal of removing all design flaw instances detected by *inFusion*.

—after all the modifications targeting the removal of a particular type of flaw were performed upon the source code we run the available tests and performed a new set of measurements.

The initial version of the system as well as all the performed refactorings are freely available for download[5]. Figure 3 summarizes the results of our measurements, by depicting the maximum values for the two measured variables (memory usage and CPU time) after each refactoring step, which involved the removal of design flaw instances of a given type.

[5]http://cs.upt.ro/∼cristina/jhotdraw-refactorings.zip

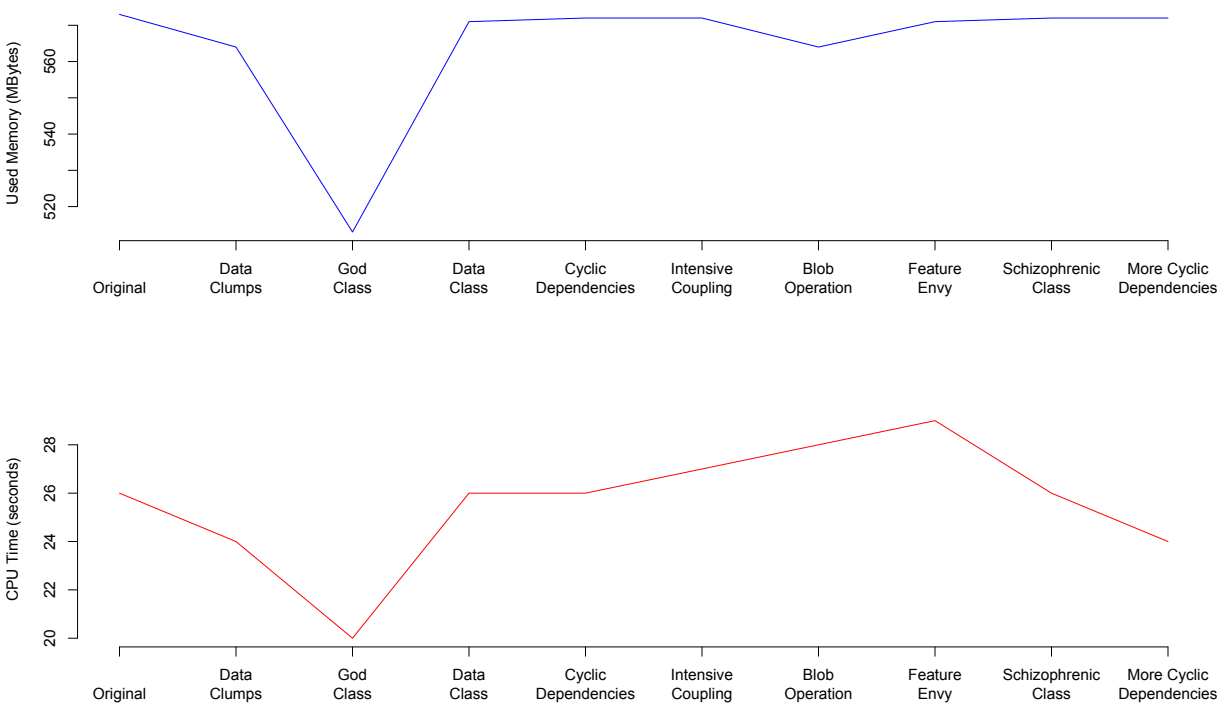


Fig. 3: Resource usage for JHotDraw.

3.2.2 *The findings.* Next we describe the findings of this incremental refactoring process. The performed measurements from Figure 3 show that:

—removing design flaws from a system has an impact towards the used computational resources.

—the necessarily amount of memory in all of the refactored versions is less when compared to the initial version of the system.

—the CPU time for running the tests is sometimes smaller when compared to the initial version of the system.

Next, based on the obtained data, we will answer the following research question: *Do the runs of the refactored source code tend to use fewer resources than the run of the initial version of the source code?*

According to the data from Figure 3, for 9 out of the 9 runs of the refactored source code the used memory is less than the used memory of the initial version of the system. We run the next statistical test: *prop.test(9, 9, 0.5, alternative="greater")* and since p-value is less than 0.05 (0.00383) we can conclude that the runs of the refactored source code where some design flaws have been removed tend to use less memory than the initial version of the system. Regarding the CPU Time, only 3 out of the 9 runs reveal lowered values; since the p-value of the statistical test *prop.test(3, 9, 0.5, alternative="greater")* is greater than 0.05 (0.7475) we cannot conclude that the refactored version tend to complete in less time than the initial version of the source code.

3.2.3 *The flaws.* According to *inFusion*, the most frequent design flaw is *Data Clumps* [Fowler et al. 1999]. This flaw consists of having the same sequence of parameters passed to different methods for many times. *inFusion* detected 38 methods with *Data Clumps*, but at a closer inspection we noticed that there are only three sequences of different parameter "clumps". Consequently, the removal consisted of creating three new classes and passing as parameter instances of those classes when appropriate.

The second refactoring addressed the removal of the single *God Class* detected in *JHotDraw*. *God Classes* tend to group unrelated pieces of functionality and to access directly non-encapsulated data members from other classes [Riel 1996]. In order to remove design flaw we split the `Geom` class into two classes, having in each class only cohesive functionality.

The next refactoring targeted the removal of *Data Classes*. In the refactoring process we increased the data-behavior locality and reduced the visibility of some public data members. However, due to the excessive complexity of the refactoring, we kept one *Data Class* as we found it very difficult to solve its dependencies to the many external classes that access it.

Next, the *Cyclic Dependencies* [Martin 2002] were removed by relocating some classes among packages. We also addressed the single *Intensive Coupling* [Riel 1996] case, exhibited by the method `getCursor()` from the `LocatorHandle` class. This method contained a large number of methods calls from the `RelativeLocator` class, as part of a complex branching structure. We refactored the method by extracting the method fragment with the many external method calls, and moving it to the class named `RelativeLocator`.

The system does also exhibit two *Blob Operations* (*i.e.*, large and complex methods [Fowler et al. 1999]). We refactored the first case (`TextAreaFigure.drawText()`) by splitting the method into some smaller methods inside the same class. In the second case (`ShortestDistanceConnector.findPoint()`) we decreased the number of the calls performed inside the refactored method by storing the values returned by the called methods into a local array whose values are interrogated.

We did also correct the *Feature Envy* flaws, which refer to methods that use heavily data from other classes instead of the data members from their definition classes [Riel 1996]. The refactoring involved moving some functionalities in the classes that provide the data these functionalities rely on. Eventually, we refactored the *Schizophrenic Classes* (*i.e.*, classes capturing more than an abstraction [Riel 1996]) mainly by splitting the classes. Last, but not least, we had to remove some additional *Cyclic Dependencies* which were involuntarily added while performing the *Feature Envy* refactoring.

4. THREATS TO VALIDITY

In this section we present the threats to validity associated to our empirical study, following the guidelines from [Yin 2002].

4.1 Construct validity

This type of threats are connected to the extent the operational measures for the concepts being studies were established correctly [Yin 2002]. Within the case study presented in this paper these threats are mainly related to the errors performed during data extraction. The possible errors are due to the extraction of (i) design entities from the source code, and (ii) measures regarding the resource usages. We consider that these threats are mitigated to a large extent as we employed a set of well-known and reliable tools.

4.2 Internal validity

This aspect of validity is related to the causal relations that are inferred. During our study we did not modify the functionalities of the analyzed systems and this is is reflected either by presenting the altered source code (for the first case study) or by passing the available suite of tests, in the case of *JHot-*

Draw. Since knowing that the functional behaviour of the system was not altered strongly depends on the quality of the existing tests, our confidence is based on the good test coverage of *JHotDraw*.

4.3 External validity

This threat concerns the possibility to generalise the provided results. The reported results are obtained by analyzing mainly a single software system. We do not suggest generalizing our research results to other systems unless further case studies are performed. We intend to replicate this study against other systems in order to see if the results obtained in this study can be generalized.

4.4 Reliability validity

This aspect concerns the fact that a later investigator that conducts the same case study like the one presented here should obtain the same results and, consequently, reach the same conclusions. We have provided all the needed information about the conducted study, and therefore we consider that the study is perfectly replicable. The source code of *JHotDraw* is freely available, as well as the refactorings we performed for removing design flaws. Also, the software tools used for extracted the presented data are properly introduced in this paper.

5. CONCLUSIONS AND FUTURE WORK

In this paper we present an empirical study performed upon two case studies (a simple one, as well as a well-known software system) providing evidence about the impact of various design flaws on the used resources of the investigated systems. We showed that design flaws increase the used amount of memory and influence the CPU time.

Our approach involved an important number of design flaws, including *Cyclic Dependencies* [Martin 2002], *Data Clumps*, *Blob Operations* [Fowler et al. 1999], *Data Classes*, *God Class*, *Intensive Coupling*, *Feature Envy*, *Schizophrenic Classes*, and *Cyclic Dependencies* [Riel 1996] in the two case studies considered.

We consider our approach as being dependent on the available tests and, consequently, a further step should be the inspection of an application actively accessing a cloud environment. In order to achieve this step, we intend to deploy an application in a real cloud environment and monitor its performance in terms of the same computing resources as considered in this work. This task could be accomplished, for example, by freely deploying the application in CloudBees[6] and perform the measurements using NewRelic[7]. Additional measurements will be performed by deploying the applications in the experimental cloud setup at the HPC center from West University of Timişoara[8].

REFERENCES

Gabriele Bavota, Bogdan Dit, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. 2013. An empirical study on the developers perception of software coupling. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 692–701. http://dl.acm.org/citation.cfm?id=2486788.2486879

Anton Beloglazov, Jemal H. Abawajy, and Rajkumar Buyya. 2012. Energy-aware resource allocation heuristics for efficient management of data centers for Cloud computing. *Future Generation Comp. Syst.* 28, 5 (2012), 755–768. http://dblp.uni-trier.de/db/journals/fgcs/fgcs28.html#BeloglazovAB12

Anton Beloglazov, Rajkumar Buyya, Young Choon Lee, and Albert Y. Zomaya. 2010. A Taxonomy and Survey of Energy-Efficient Data Centers and Cloud Computing Systems. *CoRR* abs/1007.0066 (2010). http://dblp.uni-trier.de/db/journals/corr/corr1007.html#abs-1007-0066

[6] http://www.cloudbees.com
[7] http://newrelic.com
[8] http://hpc.uvt.ro

Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, and Ioannis Stamelos. 2003. A controlled experiment investigation of an object-oriented design heuristic for maintainability. *Journal of Systems and Software* 65 (2003).

Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andr Barroso. 2007a. Power Provisioning for a Warehouse-sized Computer. In *The 34th ACM International Symposium on Computer Architecture*.

Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. 2007b. Power provisioning for a warehouse-sized computer. In *34th annual international symposium on Computer architecture*. 13–23.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Saurabh Kumar Garg, Chee Shin Yeo, and Rajkumar Buyya. 2011. Green Cloud Framework for Improving Carbon Efficiency of Clouds.. In *Euro-Par (1) (Lecture Notes in Computer Science)*, Emmanuel Jeannot, Raymond Namyst, and Jean Roman (Eds.), Vol. 6852. Springer, 491–502. http://dblp.uni-trier.de/db/conf/europar/europar2011-1.html#GargYB11

Kay Grosskop and Joost Visser. 2013. Identification of Application-level Energy-Optimizations. In *Proceedings of ICT for Sustainability (ICT4S 2013)*. 101–107.

Abram Hindle. 2012. Green mining: Investigating power consumption across versions.. In *ICSE*, Martin Glinz, Gail C. Murphy, and Mauro Pezz (Eds.). IEEE, 1301–1304. http://dblp.uni-trier.de/db/conf/icse/icse2012.html#Hindle12

ISO/IEC. 2001. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC.

Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Guéhéneuc. 2009. An Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *16th Working Conference on Reverse Engineering*.

D. Kliazovich, S.T. Arzo, F. Granelli, P. Bouvry, and S.U. Khan. 2013. e-STAB: Energy-Efficient Scheduling for Cloud Computing Applications with Traffic Load Balancing. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing*. 7–13. DOI:http://dx.doi.org/10.1109/GreenCom-iThings-CPSCom.2013.28

Dzmitry Kliazovich, Pascal Bouvry, Yury Audzevich, and Samee Ullah Khan. 2010. GreenCloud: A Packet-Level Simulator of Energy-Aware Cloud Computing Data Centers.. In *GLOBECOM*. IEEE, 1–5. http://dblp.uni-trier.de/db/conf/globecom/globecom2010.html#KliazovichBAK10

YoungChoon Lee and Albert Y. Zomaya. 2012. Energy efficient utilization of resources in cloud computing systems. *The Journal of Supercomputing* (2012), 268–280.

Wei Li and Raed Shatnawi. 2007. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of Systems and Software* 80 (July 2007). Issue 7.

Radu Marinescu and Daniel Raţiu. 2004. Quantifying the Quality of Object-Oriented Design: the Factor-Strategy Model. In *Proceedings 11th Working Conference on Reverse Engineering (WCRE'04)*. IEEE Computer Society Press, Los Alamitos CA, 192–201.

Robert Cecil Martin. 2002. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall.

Robert C. Martin. 2008. *Clean Code. A Handbook of Agile Software Craftsmanship*. PrenticeHall.

Steffen M. Olbrich, Daniela S. Cruzes, and Dag I.K. Sjøberg. 2010. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. In *IEEE International Conference on Software Maintenance*.

Ralph Peters and Andy Zaidman. 2012. Evaluating the Lifespan of Code Smells using Software Repository Mining. In *Proc. 16th European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE Computer Society.

R Development Core Team. 2010. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Web page: http: //findbugs.sourceforge.net. http://www.R-project.org ISBN 3-900051-07-0.

Arthur Riel. 1996. *Object-Oriented Design Heuristics*. Addison Wesley, Boston MA. 400 pages.

Shuaiwen Song, Rong Ge, Xizhou Feng, and Kirk W. Cameron. 2009. Energy Profiling and Analysis of the HPC Challenge Benchmarks. *IJHPCA* 23, 3 (2009), 265–276. http://dblp.uni-trier.de/db/journals/ijhpca/ijhpca23.html#SongGFC09

Mladen A. Vouk. 2008. Cloud Computing - Issues, Research and Implementations. *CIT* 16, 4 (2008), 235–246. http://dblp.uni-trier.de/db/journals/cit/cit16.html#Vouk08

Robert K. Yin. 2002. *Case Study Research: Design and Methods., 3rd edition*. SAGE Publications.

Qi Zhang, Lu Cheng, and Raouf Boutaba. 2010. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications* 1, 1 (2010), 7–18. DOI:http://dx.doi.org/10.1007/s13174-010-0007-6