Analyzing the Harmful Effect of God Class Refactoring on Power Consumption

Ricardo Pérez-Castillo and Mario Piattini,
University of Castilla–La Mancha

// Although research indicates that the refactoring field is now sufficiently mature to improve system maintainability, most refactoring techniques decrease sustainability. In particular, the excessive message traffic derived from refactoring god classes increases a system's power consumption. //

THE CONCEPT OF sustainability is the culmination of a trend that combines environmental, social, and business interests. Energy efficiency and other sustainability efforts are the norm in creating material products such as cars, light bulbs, and computer hardware, but "going green" isn't very common in software design and development.[1] Indeed, the ever-increasing usage of software products has boosted energy demand, with a 2011 study reporting that the approximate energy consumption of world datacenters jumped from 51 billion kWh in 2005 to 130 billion kWh in 2010. US and world datacenter electricity use grew by about 36 percent and 56 percent, respectively, from 2005 to 2010,[2] totaling about 1.3 percent of world electricity use and 2 percent of US electricity use in 2010.[3] Getting stakeholders to recognize environmental impacts requires building sustainability concerns into development processes—that is, having software architects, designers, and developers optimize their software products for sustainability during development.[4]

Sustainability science is described as a metadiscipline that transcends and subsumes knowledge from many other fields,[5] and part of its goal is to measure impact. The direct and indirect negative impacts from green and sustainability software's development, deployment, and usage are minimal or have a positive effect on sustainable development.[6] Therefore, the activity of defining and developing software products in such a way that the negative and positive impacts on sustainable development are continuously assessed, documented, and used helps further optimize the software product.[7]

Traditional software development cycles and methodologies emphasize software quality features such as maintainability, but to date, they haven't focused on sustainability or software's greener aspects.[1] A common practice used to improve a system's maintainability, for example, is to detect bad smells or anti-patterns in its architecture.[8] Anti-patterns document recurring solutions to common design or architectural

 0740-7459/14/$31.00 © 2014 IEEE

problems; they illustrate what not to do and how to fix a problem when you find it.[9] Any occurrences of anti-patterns are refactored to overcome their negative consequences; thus, the refactoring is a correctness-preserving transformation that improves the quality of the software without altering semantics.[10]

Most anti-pattern detection and refactoring techniques deal with improvements to the architecture in terms of maintainability. However, maintainability improvement can degrade sustainability. To assess this phenomenon, we performed a case study to detect god classes (a particular anti-pattern) as well as their refactoring opportunities. The extra power consumption we found came from the increase in message traffic derived from the architecture modification.

Sustainability vs. Maintainability

Most software improvement techniques focus on changing information system architecture to improve performance or maintainability. Our primary research question is whether the application of traditional architecture improvement techniques (in this case, design refactoring for anti-pattern detection) transform system architectures into systems that consume more power.

The advantage of applying refactoring transformations is fully justified from the maintainability viewpoint. After applying the refactoring operators, we get a modular architecture with fine-grained elements that have higher cohesion and lower coupling. But despite these benefits, the refactoring transformations might create harmful side effects in terms of sustainability. We found that when certain refactoring operators are applied in object-oriented systems (such as "extract new class" or "extract new method"), they can lead to higher power consumption via excessive message traffic between objects.

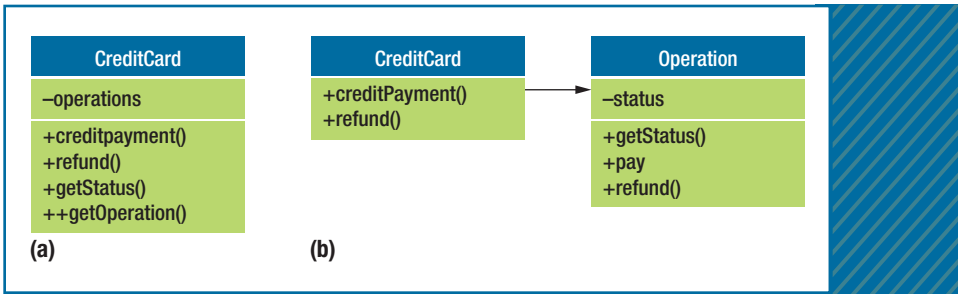

FIGURE 1. An example payment system. Comparing (a) the original god class with (b) the simple refactored solution highlights benefits and possible downsides.

Motivating Example

Imagine a system design with several god classes—the god class anti-pattern refers to classes that perform most of the heavy work in a system; other classes have minor roles. God classes can easily be recognized as single, complex controller classes (often with names containing "controller" or "manager") surrounded by simple data container classes. God classes only have accessory operations (such as `get()` and `set()`) and perform little or no computation of their own.[9]

More specifically, let's consider a payment system in which we have to make payments and refunds. Figure 1a shows a fragment of the class diagram for a possible system architecture. The CreditCard class can be considered a god class because it contains almost the whole intelligence. It retrieves each operation and checks the status (accepted or rejected) and, depending on status, performs the payment or refund. This architecture design is not very cohesive and highly coupled with data classes.

Figure 1b, on the other hand, provides a simple refactored solution in which a class for operation intelligence has been extracted. The Operation class simply reports its status (accepted or rejected) and responds to `pay()` and `refund()` invocations. CreditCard does all the work: it requests information from the Operation class, makes decisions, and tells the Operation class what to do. In this scenario, the architecture coupling has been reduced—it's more maintainable—but this new design will probably require extra messages if it is to perform an operation of the subordinate class. Hence, the message traffic between objects increases by a factor of two or more. Consequently, additional messages over several executions could lead to higher power consumption owing to the additional processor and memory resources needed to process these messages over time.

Research Hypothesis and Goal

As already mentioned, our hypothesis is that power consumption decreases as a result of reducing object message traffic. Our goal is to

demonstrate that when common refactoring patterns are applied under the detection of well-known anti-patterns, they lead to excessive object message traffic, thereby spiking power consumption. Our study's aim, which is primarily focused on god class anti-patterns, is to serve as the starting point with which to later provide refactoring transformations that take into account maintainability challenges (such as low coupling and high cohesion) and sustainability concerns (such as decreased power consumption).

> Common refactoring patterns applied under the detection of well-known anti-patterns lead to excessive object message traffic.

Research Method

To assess our hypothesis, we conducted a case study with two open source systems (http://alarcos.esi.uclm.es/per/rpdelcastillo/Experiments.html#architectureSustainability).

We chose Informa and NekoHTML from the Qualitas Corpus (http://qualitascorpus.com), a collection of software systems intended to be used for empirical studies of code artifacts. Informa (http://informa.sourceforge.net) provides an RSS (Rich Site Summary) library based on the Java platform and gives users a harmonized view of a news channel object model (such as RSS 0.9x, RSS 1.0/RDF, RSS 2.0, Atom 0.3, and Atom 1.0). NekoHTML (http://nekohtml.sourceforge.net) is a simple HTML scanner and tag balancer that lets application programmers parse HTML documents and access the information using standard XML interfaces.

We analyzed our systems under study to detect occurrences of the god class anti-pattern throughout their architectures. Specifically, we used JDeoderant (http://jdeodorant.com), an automatic Eclipse plug-in that employs a variety of methods and techniques[11] to identify code smells and suggest the appropriate refactoring operators with which to resolve them. After detecting god classes, we applied all the extracted class refactoring operators suggested by JDeodorant to obtain a refactored version of both systems, which we call InformaR and NekoHTMLR in our study.

After obtaining the refactored versions of the systems, we measured their object message traffic by using the same execution scenario to provide the same conditions. We considered a representative execution scenario by using the built-in test cases of both systems, but the execution based on test cases provided an appropriate execution scenario because it covers almost the entire functionality. To quantify the object operation invocations, we traced and profiled the source code of both systems and both versions by using the Eclipse Test and Performance Tools Platform (TPTP; www.eclipse.org/tptp).

In addition to measuring the message traffic, we measured the power consumption of both systems, as well as that of their original and refactored versions. The execution scenario was once more conformed through the test cases provided by each system's developers. Again, we executed all the systems without instrumentation to avoid the bias derived from the time spent registering the message traffic log. We performed this new execution by using a computer with a core 2 duo processor of 2.66 GHz and 4 Gbytes RAM. During execution, we measured the power consumption with an energy logger (Volcraft Energy Logger 4000), which spanned the energy plug and computer and recorded each second's energy consumption in watts (W). We also measured processor usage in parallel with power consumption.

Finally, after collecting all this data, we analyzed and interpreted the results.

Experimental Results

In total, we detected 21 of 116 classes as god classes in Informa and 10 of 60 in NekoHTML, or 18 percent and 17 percent of the total number of classes, respectively (see Table 1). Of this amount of god classes, we detected 49 and 26 extractable classes, respectively. After detecting the god classes, we performed all the suggested refactoring transformations (extract classes) to get 11 and 14 new classes in both systems. The number of new extracted classes was lower than the number of god classes because most of the suggested extractable classes overlapped the various god classes.

Table 1 provides most of the relevant architectural/design metrics of the original and refactored system under study, in addition to the difference between both versions. It also shows that InformaR has 11 new classes, which entail 9.48 percent more than the original system, while NekoHTMLR contains 23.33

TABLE 1

Architectural metrics, message traffic, and power consumption during execution.

	Measure	Informa	InformaR	Difference (%)	NekoHTML	NekoHTMLR	Difference (%)
Architecture	Lines of code	9,739	9,891	1.56	7,938	8,179	3.04
	Classes	116	127	9.48	60	74	23.33
	Methods	996	1,024	2.81	473	523	10.57
	Afferent coupling	10	10.5	5.00	5.29	5.43	2.71
	Efferent coupling	7.21	7.57	4.95	5.57	7.29	30.78
	Cycle complexity	1.87	1.84	–1.18	3.44	3.23	–6.24
Refactored	God classes	21	0		10	0	
	Ratio of god classes (%)	18.1	0		17	0	
	Extracted classes	49	0		26	0	
Execution	Test cases	337	337	0.00	4,201	4,201	0.00
	Errors	71	71	0.00	0	0	0.00
	Failures	18	19	5.56	1,800	2,200	22.22
	Messages	6,221	97,846	1,473	1,550,848	7,900,600	409
	Time (s)	57	60	5.26	22	27	22.73
Power	Total watts	2,052.6	2,207.7	7.56	743.9	893.4	20.10
	Watts/s	36.7	37.4	1.91	33.8	34.4	1.62

percent more classes. These new classes are related to the respective increase of 152 and 241 LOC, which represent an increase of 1.56 percent and 3.04 percent. As a result of extracting fine-grained classes from the god classes during refactoring, the afferent and efferent coupling (CA and CE) are worse (around 5 percent in the case of InformaR and between 2.7 percent and 30.7 percent for the afferent and efferent coupling of NekoHTMLR). However, as expected, the complexity was reduced by 1.18 percent and 6.24 percent for each system, respectively.

After refactoring the original system, the architectures of both systems would appear to be more maintainable according to the aforementioned measures and owing to

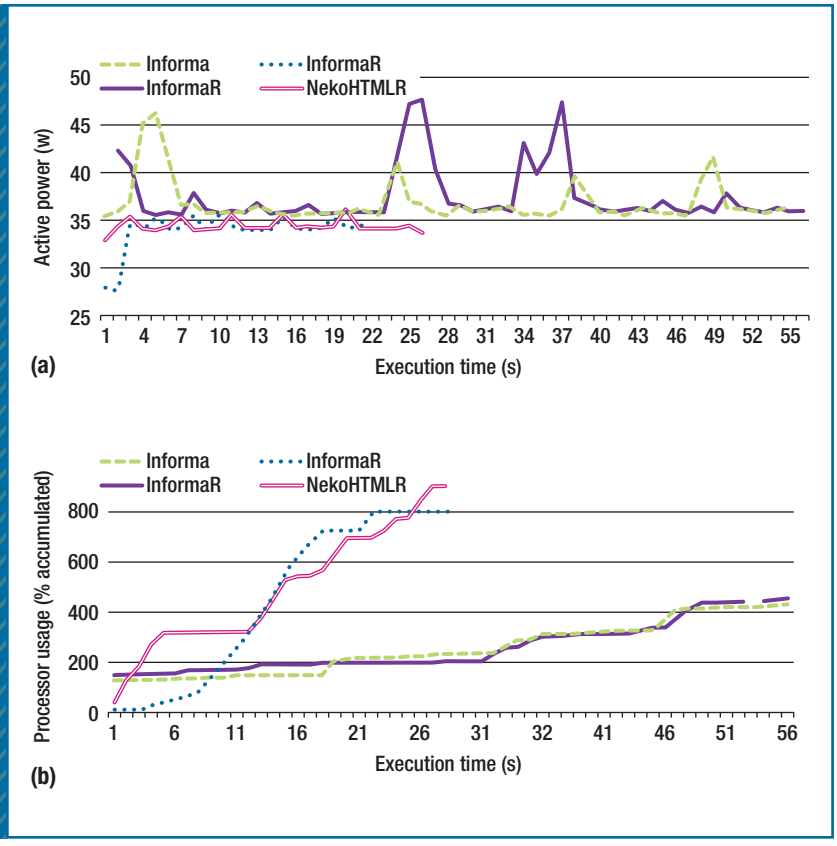

FIGURE 2. A comparison. (a) Power consumption and (b) accumulated processor usage during execution indicate more power consumed with more messages.

the fact that we dealt with all the occurrences of god classes. Next, we had to measure message traffic. We executed instrumented versions of both systems throughout the test suite incorporated in Informa and NekoHTML source code, and then we analyzed the two invocation logs recorded during that execution and quantified the messages. In total, 6,221 messages were interchanged during Informa's execution, and 97,846 messages were produced in the refactored version. This dramatic increase was 14 times greater because of the extra calls between the additional fine-grained classes extracted during refactoring. For NekoHTML, the increase was 4 times greater: message traffic varied between 1,550,848 and 7,900,600. The difference between both systems lies in the fact that god class refactoring might lead to new inheritance or composition relations, each of which could lead to different message traffic. Inheritance and delegation is probably one of the factors for different message traffic values in the legacy systems we analyzed.

After demonstrating that the refactored architecture produces between 4 and 14 times more messages for both systems under study, the last step was to evaluate the power consumption and link message traffic with power consumption. Figure 2a presents the active power consumption evolution (in watts) during the execution of both systems and both versions. The peaks in the plot come from temporal increases in processor usage: certain test cases require additional computational resources.

Informa's power consumption was on average 36.7 watts, while InformaR's was on average 37.4 watts. The difference in power consumption was on average 1.91 percent. For NeckoHTML, the original system consumed 33.8 watts per second, and for NeckoHTMLR, 34.4 on average. This signifies an increase of 1.62 percent. Because execution time was higher for the refactored systems, the increases in power consumption (in terms of absolute values) were 7.6 percent and 20.1 percent, respectively. The different increases are probably due to each system's execution scenarios (which are based on test cases and could affect other areas controlled by inheritance and delegations caused by refactoring).

Figure 2b shows the accumulated percentage of processor usage during the execution of both systems. The lines follow a similar trend, but there's a processor usage spike between the lines. The increase in processor usage proved to be in line with the increase in power consumption during the execution of InformaR and NekoHTMLR.

Ultimately, while the message traffic was 14 times higher for InformaR, the power consumption only increased by around 8 percent, whereas although the message traffic was 4 times higher for NekoHTMLR, the power consumption increased by 20 percent. A better architecture in terms of maintainability can certainly be worse in terms of power consumption.

Study Limitations

The study of maintainability and sustainability is a huge endeavor because it aims to compare an internal (maintainability) and an external (power consumption) software quality. By addressing the former, refactoring does decrease software complexity, which is a desirable software architecture quality, potentially at design time; power consumption, meanwhile, is a quality at runtime. Hence, the trade-offs between the two qualities must be carefully analyzed.

Although our experimental results have shown that an architecture in which god classes have been refactored can worsen in terms of power consumption, the study has some limitations that should be mentioned. The first concerns the system under study. Despite the fact that Informa and NekoHTML are real-life open source systems, which helps ensure our study's repeatability, the study should also include additional information systems. In fact, higher and more complex systems should be analyzed under the same conditions to provide stronger results.

The second limitation was the measurement of power consumption using the energy logger, which recorded the computer's total power consumption—so power consumption wasn't just execution of Informa or NekoHTML, but of the operating system and other computer applications. Generally speaking, environmental/green performance is a pervasive quality that's difficult to measure because it's affected by every aspect of the design and execution environment. To mitigate this, we executed the computer in safe mode, with just the essential services and no additional applications executed in parallel. Other studies have, however, researched energy usage attribution—for example, the Software Energy Footprint Lab.[11] Moreover, hardware architecture plays an important role because power consumption could be related to the software's suitability to run on a specific piece of hardware. Consequently, a green system will most likely require the co-design of hardware and software, or at least a choice of hardware that's suitable for the software architecture chosen.

Another important threat to our study's validity is our usage of JDeodorant to detect anti-patterns without questioning the results provided by this tool or evaluating its false positives. Alternative tools with which to detect bad smells and anti-patterns could be used to obtain comparative results. However, JDeodorant is an open source tool that offers support and has been used in industrial refactoring projects, plus it also applies the well-proven Eclipse refactoring operators rather than implementing its own.

Similarly, the power consumption influence regarding many other patterns and anti-patterns together with additional refactoring operators should be analyzed. For example, Feature Envy is devoted to changing classes that excessively use methods from other classes. In this case, called methods are encapsulated in the envy class, but the message traffic is almost the same. However, the refactoring of long methods can sometimes lead to an increase in message traffic because it's refactored to shorter methods, and the power consumption might therefore be higher.

The last limitation is related to the execution scenario we selected, which was based on the test suite incorporated by the system developers themselves. We did this because of the impossibility of defining representative execution scenarios based on the whole functionality of the system under study. However, in future replications, the opinion of experts could be taken into account to define accurate and complete execution scenarios of the systems under study.

> Green performance is difficult to measure because it's affected by every aspect of the design and execution environment.

Despite the aforementioned limitations and necessary future work, the main implication of our preliminary study is that alternative refactoring techniques should be proposed to simultaneously achieve both maintainable and sustainable architectures. In fact, the conciliation between maintainability and sustainability might be extremely difficult, as we've shown in our motivating example, but we believe that our study is a good starting point for future research into alternative refactoring transformations.

Architects typically work on maintainability for specific business drivers or stakeholders, so some barriers with which to change their business strategy and address (slightly) higher power consumption could

ABOUT THE AUTHORS

RICARDO PÉREZ-CASTILLO is an assistant professor at the University of Castilla–La Mancha and belongs to the Alarcos Research Group at UCLM. His research interests include architecture-driven modernization, model-driven development, and business process archeology. Pérez-Castillo has a PhD in computer science from the University of Castilla–La Mancha. Contact him at ricardo.pdelcastillo@gmail.com.

MARIO PIATTINI is a full professor at the University of Castilla–La Mancha. His research interests include software quality, metrics, and maintenance. Piattini has a PhD in computer science from the Technical University of Madrid and leads the Alarcos Research Group at UCLM. Contact him at mario.piattini@uclm.es.

exist. Questions about the costs that an average CIO can save when using a sound sustainability strategy on a corporate IT landscape should therefore be answered. A more business-focused study could be carried out to build a clear organizational case for a proper sustainability strategy.

Acknowledgments

This work was supported by the R&D project GEODAS-BC (TIN2012-37493-C03-01) funded by Ministerio de Economía y Competitividad and FEDER.

References

1. S. Agarwal, A. Nath, and D. Chaudhury, "Sustainable Approaches and Good Practices in Green Software Engineering," *Int'l J. Research and Reviews in Computer Science*, vol. 3, no. 1, 2012, pp. 1425–1428.
2. G. Scanniello et al., "Using the GPU to Green an Intensive and Massive Computation System," *Proc. 17th European Conf. Software Maintenance and Reengineering* (CSMR 13), 2013, pp. 384–387.
3. J. Koomey, "Growth in Data Center Electricity Use 2005 to 2010," Analytics Press, 2012; www.analyticspress.com/datacenters.html.
4. S. Naumann et al., "The GREENSOFT Model: A Reference Model for Green and Sustainable Software and Its Engineering," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 4, 2011, pp. 294–304.
5. J.R. Mihelcic et al., "Sustainability Science and Engineering: The Emergence of a New Metadiscipline," *Environmental Science and Technology*, vol. 37, no. 23, 2003, pp. 5314–5324.
6. M. Dick, S. Naumann, and N. Kuhn, "A Model and Selected Instances of Green and Sustainable Software," *What Kind of Information Society? Governance, Virtuality, Surveillance, Sustainability, Resilience*, J. Berleur, M. Hercheui, and L. Hilty, eds., Springer, 2010, pp. 248–259.
7. M. Dick and S. Naumann, "Enhancing Software Engineering Processes towards Sustainable Software Product Design," *Proc. 24th Int'l Conf. Informatics for Environmental Protection* (EnviroInfo 2010), 2010, pp. 706–715.
8. W.H. Brown, R.C. Malveau, and T.J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons, 1998.
9. C.U. Smith and L.G. Williams, "Software Performance Antipatterns," *Proc. 2nd Int'l Workshop Software and Performance*, 2000, pp. 127–136.
10. T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Trans. Software Eng.*, vol. 30, no. 2, 2004, pp. 126–139.
11. M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice*, Springer, 2005.
12. M.A. Ferreira et al., "SEFLab: A Lab for Measuring Software Energy Footprints," *Proc. 2nd Int'l Workshop Green and Sustainable Software*, 2013, pp. 30–37.

ADVERTISER INFORMATION • MAY/JUNE 2014

Advertising Personnel

Marian Anderson: Sr. Advertising Coordinator;
Email: manderson@computer.org; Ph: +1 714 816 2139 | Fax: +1 714 821 4010

Sandy Brown: Sr. Business Development Mgr.
Email sbrown@computer.org; Ph: +1 714 816 2144 | Fax: +1 714 821 4010

Advertising Sales Representatives (display)

Central, Northwest, Far East: Eric Kincaid
Email: e.kincaid@computer.org; Phone: +1 214 673 3742; Fax: +1 888 886 8599

Northeast, Midwest, Europe, Middle East: Ann & David Schissler
Email: a.schissler@computer.org, d.schissler@computer.org
Ph: +1 508 394 4026; Fax: +1 508 394 1707

Southwest, California: Mike Hughes
Email: mikehughes@computer.org; Ph: +1 805 529 6790

Southeast: Heather Buonadies
Email: h.buonadies@computer.org; Ph: +1 973 585 7070; Fax: +1 973 585 7071

Advertising Sales Representatives (Classified Line, Jobs Board)

Heather Buonadies
Email: h.buonadies@computer.org Ph: +1 973 304 4123; Fax: +1 973 585 7071