CEPIS UPGRADE

The European Journal for the Informatics Professional
http://cepis.org/upgrade

Vol. XII, issue No. 4, October 2011

Monograph
Green ICT: Trends and Challenges
(published jointly with Novática*)
Guest Editors: *Juan-Carlos López-López, Giovanna Sissa, and Lasse Natvig*

* This monograph will be also published in Spanish (full version printed; summary, abstracts, and some articles online) by **Novática**, journal of the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*) at <http://www.ati.es/novatica/>.

CEPIS UPGRADE is the European Journal for the Informatics Professional, published bi-monthly at <http://cepis.org/upgrade>

Publisher
CEPIS UPGRADE is published by CEPIS (Council of European Professional Informatics Societies, <http://www.cepis.org/>), in cooperation with the Spanish CEPIS society ATI (*Asociación de Técnicos de Informática*, <http://www.ati.es/>) and its journal Novática

CEPIS UPGRADE monographs are published jointly with Novática, that publishes them in Spanish (full version printed; summary, abstracts and some articles online)

CEPIS UPGRADE was created in October 2000 by CEPIS and was first published by Novática and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies)

CEPIS UPGRADE is the anchor point for UPENET (UPGRADE European NETwork), the network of CEPIS member societies' publications, that currently includes the following ones:
• inforewiew, magazine from the Serbian CEPIS society JISA
• Informatica, journal from the Slovenian CEPIS society SDI
• Informatik-Spektrum, journal published by Springer Verlag on behalf of the CEPIS societies GI, Germany, and SI, Switzerland
• ITNOW, magazine published by Oxford University Press on behalf of the British CEPIS society BCS
• Mondo Digitale, digital journal from the Italian CEPIS society AICA
• Novática, journal from the Spanish CEPIS society ATI
• OCG Journal, journal from the Austrian CEPIS society OCG
• Pliroforiki, journal from the Cyprus CEPIS society CCS
• Tölvumál, journal from the Icelandic CEPIS society ISIP

Cover page designed by Concha Arias-Pérez
"Luminous Recharge" / © ATI 2011
Layout Design: François Louis Nicolet
Composition: Jorge Llácer-Gil de Ramales

Editorial correspondence: Llorenç Pagés-Casas <pages@ati.es>
Advertising correspondence: <info@cepis.org>

Subscriptions
If you wish to subscribe to CEPIS UPGRADE please send an email to info@cepis.org with 'Subscribe to UPGRADE' as the subject of the email or follow the link 'Subscribe to UPGRADE' at <http://www.cepis.org/upgrade>

ISSN 1684-5285

Monograph of next issue (December 2011)
"**Risk Management"**

Good, Bad, and Beautiful Software – In Search of Green Software Quality Factors

Juha Taina

New emerging areas of software development and usage are green software engineering and software engineering for the planet. In green software engineering, software is developed, executed, and maintained in as environment friendly a way as possible. In software engineering for the planet, software actively helps in the fight against climate change. Currently, we lack good factors for estimating how good green software is. We consider this problem on three levels. First, a developer's approach is to calculate the carbon footprint of software development, delivery, and maintenance. These factors are relatively easy to estimate but they lack an execution phase. Second, an engineer's approach is to count how much resource software uses relative to its tasks. These factors are good for green software engineering but not enough for software engineering for the planet. Finally, a planet's approach is to calculate how much software offers or how much damage it does to its environment in its life-time. Together these would give an ultimate solution for the green software factor dilemma but unfortunately it is not at all clear how to define metrics for such factors.

Keywords: Green Metrics, Green Software, Software Engineering for the Planet.

Author

Juha Taina is a university lecturer at University of Helsinki, Faculty of Science, Finland. He is an experienced teacher and researcher in software engineering. His current research interests include green software, green software engineering, sustainable development and software engineering education. <taina@cs.helsinki.fi>

1 Introduction

Software can and will help in our pursuit to sustainable development and reduction in carbon dioxide emissions. It will play an ever increasing role in controlling systems, optimising algorithms and generating alternatives for carbon intensive processes.

Software systems are present in practically every aspect of the world around us. This implies that software is present in a myriad of ways. We measure software quality with *software quality factors* that describe how software behaves in its system. In the future, we will need new factors, *green quality factors*, to define how software supports sustainable development and our fight against climate change.

From the sustainable development and climate change point of view, good software helps to reduce greenhouse gases, waste, and resource requirements while bad software increases them. Sometimes software can be downright ugly: badly written, difficult to use, and resource intensive. On the other hand, we can have elegantly beautiful software that not only reduces resource use but also affects how people see sustainable development.

In order to understand how good, bad, or beautiful software is, we need relevant metrics. Once we have such metrics we can compare software either with other software or against an absolute scale. Unfortunately, it is not at all clear what sort of metrics should be used. It is even arguable whether software has a role in sustainable development or how large the role is.

In principle it is easy to say how green a *software system* is. First we take the initial state of a system domain without the system and calculate the resources required in the domain. Then we re-calculate the required resources after we have installed the software system. If the new sum is smaller than the old one, the software system supports sustainable development. We can compare two software systems in the same way.

For instance, let us have a teleconferencing software system that allows people to use fast network connections for on-line meetings. The original domain would require everyone to travel to their meetings which would of course require resources. The new domain would cut travelling but perhaps add some extra resource requirements such as network usage. However, the net effect would most probably cut carbon emissions and required resources. The system would help in sustainable development.

The scenario gets more complex when we talk about software alone. Software is a necessary component of a software system, but so is hardware. It is not at all clear how to define how much of the resources saved is due to good software and how much is hardware-related.

In this paper we give a classification for green quality factors and define green metrics. Due to the inexact nature of the field, our definitions are neither complete nor abso-

“Software can and will help in our pursuit to sustainable development and reduction in carbon dioxide emissions”

“ A larger field called *Software Engineering for the Planet (SEP)* addresses issues and questions on where software and software engineering can help sustainable development and a carbon-free environment ”

lute. It is possible to define a different set of green quality factors, however, as far as we know this paper is the first attempt to define them. Perhaps in a few years we will have a large set of well-defined green quality factors from which we can pick the best ones.

The rest of the paper is organised as follows. Section 2 introduces green software and green software engineering. Section 3 presents our framework for green quality factors. Section 4 introduces green quality factors and metrics that are related to software development. Section 5 introduces factors and metrics related to software execution. Section 6 introduces factors and metrics related to software effects in sustainable development. Section 7 summarises and concludes the paper.

2 Background

When people think about green software, most of them are thinking about *Green IT.* In short, it is the study and practice of using computing resources efficiently [1]. Green IT is a huge field that includes everything in a software system life cycle from hardware manufacturing to minimising disposable waste. Resource usage is not a small matter in IT. For instance, the TOP500’s leading supercomputer, the K computer, consumes enough energy to power nearly 10,000 homes and costs $10 million a year to operate [2]. Each PC in use generates about a ton of carbon dioxide a year [3].

Green IT addresses the research and practice of how to reduce carbon emissions and other resource requirements of software systems. It is an important field where green software plays a remarkable role, yet "only" 2% of global carbon emissions are *directly* due to IT systems [4]. Green software can help to *indirectly* reduce carbon emissions, waste, and resource usage.

A larger field called *Software Engineering for the Planet (SEP)* [5] addresses issues and questions on where software and software engineering can help sustainable development and a carbon-free environment. While Green IT belongs to SEP, it includes other areas as well. In a recent workshop, the participants defined that SEP consists of at least the following [6]:

1. Software support for green education. Use software to support education and general knowledge about sustainable development and climate change.

2. Green metrics and decision support. Software processes and tools for environmental-friendly software design, implementation, usage, and disposal.

3. Lower IT energy consumption. Software to support or be part of Green IT.

4. Support for better climate and environment models. Let environmental scientists do their research with better software.

The need to greatly reduce carbon emissions and eventually become a carbon-neutral sustainable society is a difficult task that needs cooperation from all research fields. Software engineering is no exception.

3 Green Quality

Software quality is defined via a set of software quality factors. If software fulfils the defined factors, it is considered a quality product. A software quality factor defines its relevant requirements. If software fulfils the requirements, it fulfils the defining software quality factor. Several models of software quality factors have been suggested in software engineering literature. Galin [7] lists some of the most common classic models while new models are constantly emerging. For instance, in a recent article Naumann et al. defined sustainable-related green metrics based on the direct and indirect effects of software [8].

Any defined software quality metric is related to one or more quality factors. Hence, in order to define a new quality metric one first needs to decide what quality factors its measurements will model.

While the list of defined quality factors is comprehensive, they all lack factors for software greenness and sustainability. We need factors to define how environment friendly software is and how it will support sustainable development. We call such factors green software quality factors or simply *green factors.*

A green factor defines properties that green software must fulfil. It needs one or more *green metrics* which measures the factor fulfilment in software. For instance, if we have a green factor *stillness* (moving requires resources) we could have a metric *steps* to calculate how many steps a software developer needs to take during a software development phase. The resulting number of steps needs to be interpreted in order to decide how the development process has fulfilled the stillness factor.

Any green factor needs to fulfil one or more green requirements. The higher-level the requirements we have, the higher-level the green factors need to be defined. We require green software to fulfil three abstract requirements:

1. The required software engineering processes of software development, maintenance, and disposal must save resources and reduce waste.

2. Software execution must save resources and reduce

waste.

3. Software must support sustainable development.

These most abstract requirements give us three abstract green factors:

1. Feasibility. How resource efficient it is to develop, maintain, and discontinue software.

2. Efficiency. How resource efficient it is to execute software.

3. Sustainability. How software supports sustainable development.

The required resources can be material, energy, or human resources. For instance, *time* is a very important resource that should not be wasted. Green software is developed in the minimum possible time but not any faster.

Resource efficiency minimizes *waste*. Hence, green software minimises the waste produced both in development and execution. For instance, c*arbon dioxide* is a typical waste that must be minimised.

Software is always part of a software system. One aspect of sustainability is how software helps its system to reach its goal. Green software supports its system at maximum efficiency. For instance, the goal of an eShop system is to sell products efficiently via the web. Good eShop software supports this by offering a positive user experience.

It is important to notice the difference between green software and a *green software system*. We can have green software in a non-green software system and vice versa. For instance, a coal power plant software system is not green but optimising software that minimises carbon dioxide emissions from the coal power plant software system is green. It supports sustainable development when it minimises emissions and it supports its system to reach the goal of producing maximum energy with minimum waste.

4 Feasibility

The software life cycle begins when the decision to build software is made. Green software is built with green methods. Feasibility defines how projects and processes that develop, maintain, and discontinue software follow sustainable development.

Feasible software is built with sustainable processes. Its software engineering processes support sustainable development, minimise resource requirements and produce minimal waste.

The emphasis on feasibility is in software engineering processes instead of final products. Fortunately, we know how to measure processes and have good factors and metrics for it. We can easily use the same metrics as in regular industry. Our feasibility factors are related to human behaviour instead of software execution. For instance, it is possible to calculate how many kilometres an average developer needs to travel during the development phase. The result is a measurable absolute value that can be interpreted to required resources and produced waste.

In an earlier paper, we calculated development carbon footprints to analyse feasibility [9]. It is relatively straightforward to calculate required resources for various software life cycle steps such as development, maintenance, and disposal. It is relatively easy to calculate how many human resources are needed to manage software.

Since feasibility is an easy factor to measure, it is a practical factor to compare software. In other words, we can use feasibility to advertise our software and especially our software development and management processes: "*Our software is good because feasibility measurements show optimal development processes*."

Here are a few good feasibility metrics. They are based on traditional project and process metrics.

1. A Carbon Footprint (CF) defines how much carbon dioxide a software development, management, or maintenance phase will emit. This is the most important green metric and eventually all green factors should be calculated with it. In the case of the feasibility factor, its values are relatively easy to calculate. In the context of other factors, it may give more ambiguous results.

2. Energy defines how much energy is consumed during the development phases. This metric does not care how energy is produced. The CF metric does that.

3. Travel defines how much travelling time is required during the development phases. This is an important metric because travelling takes time and requires resources. The less travelling that is required during the development phases, the better the feasibility. This is true in cases when travelling is seemingly emission-free, like when walking or using a bicycle.

4. Waste defines how much resources are consumed in activities that create no visible value to software end users. Waste can be physical, energy, or process waste. Process waste defines operations in a process that require resources but do not produce anything valuable. For instance, idle development times and waiting queues are waste because they do not create any value.

The CF metric is the most important of the listed metrics. It is a direct metric that can be used as a basis for various indirect metrics. Is software A more feasible than software B? Calculate functionality/CF.

Based on our earlier paper it appears that "travelling rules" in development resource requirement analysis [9].

“Feasible software is built with sustainable processes. Its software engineering processes support sustainable development, minimise resource requirements and produce minimal waste”

Fortunately travelling is one of the easiest areas to reduce. A lot of travelling can be replaced with conference meetings using software, online discussions, and shared development screens.

5 Efficiency

Good software does not waste resources whether from its system or from its users. Bad software steals time and increases waste. Efficiency defines how software behaves when it comes to saving resources and avoiding waste.

We define the following factors and metrics for efficiency. With these factors it is possible to estimate how software reduces waste.

1. *CPU-intensity*. How many CPU cycles software consumes. A good metric is *cycle count.*

2. *Memory usage*. How much memory is used and how. A good metric is *main memory consumption.*

3. *Peripheral intensity*. How much peripheral equipment is used. A good metric is *peripheral usage time* that can be further translated to carbon dioxide emissions or other waste metrics.

4. *Idleness*. How much software is idle. A good metric is *idle time*. This factor and metric are relevant to only certain types of software systems such as virtual servers.

5. *Reflectivity.* How software indirectly affects its domain. Good reflectivity metrics need more research.

The most intuitive software resource is a CPU cycle. Each executed cycle requires energy and hence has a measurable waste value. We can measure it and translate it to any environment based metric such as the amount of carbon emissions. We call this *CPU-intensity*. It defines how many CPU cycles software consumes.

While CPU-intensity is a possible metric for efficiency, it is a bit limited and difficult to calculate. Software requires more than just CPU cycles. It needs main memory and peripherals in order to execute. Hence, we need metrics for main memory and peripheral requirements.

The peripheral requirements for software are easier to estimate. We can calculate how many requests software generates for peripherals and what resources are required. A simple metric is to calculate energy requirements for a peripheral and partition the measures to all software that requires services from it. We call this *peripheral intensity.* It defines how much peripheral equipment software requires.

The main memory requirements are more difficult to estimate. We need an attribute to evaluate how much memory is in use and how much it required resources. We call this *main memory consumption*. It defines how software uses main memory.

In case of main memory, it not only matters how much data is stored but also what kind of data is stored. For instance, in a very low-energy environment it is useful to minimise memory consumption and reset unused bytes to nil. The result is a minimal number of set bits in main memory. We save energy because it is more efficient to refresh a reset bit than a set bit.

High CPU-intensity, peripheral intensity, and main memory usage are justified if they are for a good cause. Software can consume resources as long as it helps its software system to reach its goal efficiently. This is especially true of CPU intensity. Consumed CPU cycles are usually useful. Idle cycles, on the other hand, are never useful.

Any idle cycle of a CPU is a wasted resource. It uses energy and creates waste but does not give anything back. A typical home desktop or even an IT server can be 90% idle [1]. Such a system is like a plane with 90% empty seats. Fortunately Green IT has already acknowledged the problem and with proper IT-server solutions it is possible to minimise idle cycles.

So far we have analysed the resource requirements for *inputs* to software. They are the requirements for software to execute. However, in efficiency the resource requirements for *outputs* from software are also meaningful: how much software usage indirectly affects required resources.

“ Even if we are not yet able to define good absolute metrics for all factors, we think that recognising green software factors is already an important contribution ”

A typical output of desktop software is a report. If the report is viewed on a display, its resources are included in the peripheral resources. If it is printed, paper and ink usage should be included. If it is sent to several people, the resource requirements to process the output should be included for all receivers. We call this *reflectivity:* how much software affects others and how.

Reflectivity is perhaps the most important factor in efficiency. The amount of generated waste can be devastating. For instance, consider reflectivity of spamming software.

There is one reflectivity aspect that may not be as obvious as the others. On laptop and desktop computers in particular, a special type of software has a nasty indirect way to increase waste without being CPU-intensive: memory resident software. It does not use much resources but when it does it is always at a critical time: when the host computer is turned on. They slow the computer start-up because that is when they are automatically loaded. The slower the start-up, the higher the probability that end users would rather leave the computer on than turn it off. This directly weakens software efficiency. Hopefully current trend to forwards flash drives and optimised hibernation algorithms allows computers to hibernate more efficiently. According to Lo et al., the rebooting of a hibernated system with a flash drive can be about one second [10]. Such reboot times would effectively nullify this aspect.

These factors and their metrics are not as intuitive as the feasibility software metrics. The metrics are difficult or almost impossible to calculate, yet it is clear that green software needs to execute in a green way. The less resources software consumes and wastes directly or indirectly, the greener is its execution.

6 Sustainability

So far we have discussed green software factors that are directly or indirectly related to software itself. However, perhaps apart from reflectivity, such factors do not tell much about the possible environmental effects of software.

We would like to have factors with easily measurable metrics that would answer the following questions:

1. How fit for purpose software is for its system?

2. How software supports its system waste reduction in the system domain?

3. What is the value of software in sustainable development?

We can analyse the requirements from two points of view:

1. Software as part of a software system, and

2. Software as a stand-alone product.

From the software system point of view software is an indivisible component. In that case sustainability can be defined only within a software system. This in turn implies that sustainability is a relative factor. We cannot compare sustainability of two pieces of software unless they are in similar software systems.

While reducing sustainability to similar systems is a serious simplification, it gives us simple tools to analyse sustainability within a software system. We can define sustainability as having the following factors:

1. Fit for purpose defines how software helps its system to reach its goal.

2. Reduction defines how software supports its system in waste reduction.

3. Beauty defines the value of software in sustainable development.

All these factors can be measured similarly inside a software system:

1. Select a reference system.

2. Calculate whatever metric you want with the reference system.

3. Calculate the same metric with the software system.

4. Compare results.

For instance, if we have software from a car brake system that saves 10% of braking energy, the reduction of this software is 10%. If the estimated maximum energy saving of the braking system is 20%, software fit for purpose is -10%. If we have software that increases global awareness of climate change by 15%, software beauty is 15%. All these values are relative to the software system where software is executed.

Unfortunately, it is not possible to have absolute metrics with these factors. The factors are subjective and depend on the system where software is executed and the domain where the system is used. In order to measure these factors, we need to be domain-specific. For each system, we need to define a reference system with which to compare. Then we can calculate how much the new system saves when compared with the old one.

The analysis becomes more complicated when we compare software in different systems. In fact, software as a stand-alone product might get unintuitive results from the previous factors.

For example, in an earlier paper we had software that helped to reduce carbon plant emissions by 10% [11]. The fit for purpose of the software is good since it obviously helps the plant to produce energy more efficiently than with a reference system (without the software). The reduction of the software is good since it helps the plant to reduce carbon emissions. The beauty of the software is good since by reducing carbon emissions it supports sustainable development. Hence, we can say that the carbon plant software is green.

In the same paper, we had an example of software that controls a pack of windmills. The software does its work but not very efficiently. Because of this, the windmills do not work at maximum efficiency. Obviously, the fit for purpose of the software could be better. The windmills are not as efficient as they could be. However, the fit for purpose of the windmill depends on the reference system. If the software helps the windmills to create more energy (but not at maximum level), the software is purposeful. The reduction of software is probably neutral since it neither helps to reduce emissions nor helps to cut them. (A windmill has carbon dioxide emissions from physical components and maintenance.) The beauty of the software is not good since *software* does not support sustainable development (but the software *system* indeed does). With our definitions, we can say that windmill software is not green.

The result may be unintuitive at first, because there is an underlying assumption that windmills are always more environment friendly than coal power plants. However, here software systems differ from each other. With our definitions, we can only compare coal power plant software with similar software and windmill software with similar software. Our power plant software is green because it cuts emissions compared with a reference power plant without control software.

If we want factors with more intuitive metrics, we need to measure software as a stand-alone product. With abstract factors, such metrics would also be abstract. Software can be anything and exist everywhere. As a minimum, we need to define a problem domain where software is executed before we can define sustainability. For instance, we can compare windmill software and power plant software since their common domain is energy production. We cannot compare car brake software with eShop software since they do not have a common problem domain. Nevertheless it is unclear currently how to get absolute beauty measurements of software as a stand-alone product.

7 Conclusion

In this paper we have defined green software via green software factors. Of the defined factors, feasibility is the most intuitive and easiest to measure. It addresses software development projects, processes, and methods. Efficiency is relatively intuitive but may not be easy to measure. The interpretation of efficiency metrics is not straightforward. Sustainability is the most abstract factor and the most difficult to measure. We can get relative measures by comparing software within a software system but that may give us unintuitive results. We wish to measure software sustainability without a software system but currently we lack metrics to do it.

Even if we are not yet able to define good absolute metrics for all factors, we think that recognising green software factors is already an important contribution. It matters how we design, implement, manage, and discontinue software in sustainable development. It matters what purpose software serves directly and indirectly. It matters how software supports sustainable development and waste reduction. We have tools and techniques to create and execute software that can be truly beautiful.

References

[1] J. Lamb. "The greening of IT. How companies can make a difference for the environment". IBM Press, 2009.
[2] T. Geller. "Supercomputing’s exaflop target". Communications of the ACM 54,8 (2011), pp. 16-18.
[3] S. Murugesan. "Harnessing green IT: principles and practices". IT Professional 10,1 (2008), pp. 24-33.
[4] Gartner Newsroom. <http://www.gartner.com/it/page.jsp?id=503867>.
[5] Software engineering for the planet, <http://groups.google.com/group/se-for-the-planet>.
[6] First International Workshop on Software Engineering and Climate Change. Orlando, FL, 2009, <http://www.cs.toronto.edu/wsrcc/WSRCC1/index.html>.
[7] D. Galin. "Software quality assurance: from theory to implementation". Pearson Education Limited, 2004.
[8] S. Naumann et al. "The GREENSOFT model: a reference model for green and sustainable software and its engineering". Sustainable Computing: Informatics and Systems, 2011.
[9] J. Taina. "How green is your software?". Proceedings of the First International Conference of Software Business (ICSOB 2010). Jyväskylä, Finland, 2010, pp. 151-162.
[10] Shi-wu Lo, Wei-shiuan Tsai, Jeng-gang Lin, Guanshiung Cheng. "Swap-before-hibernate: a time efficient method to suspend an OS to a flash drive". Proceedings of the 2010 ACM Symposium on Applied Computing (SAC ’10). ACM, New York, NY, USA, pp. 201-205.
[11] J. Taina, P. Pohjalainen. "In search for green metrics". Workshop on Software Research and Climate Change. Orlando, FL, 2009. <http://www.cs.toronto.edu/wsrcc/Taina-WSRCC-1.pdf>.