Sustainability Knowledge about Software Parts in Software Engineering Processes

Discussion of an “Information Leaflet” Approach

Benno Schmidt
Bochum University of Applied Sciences
Bochum, Germany
benno.schmidt@hs-bochum.de

***Abstract*—The ecological and sustainable design, implementation and configuration of software is a rather difficult task these days. Practical guidelines and helpful hints to assist software architects, programmers, and operators in their daily business are still rare. This paper discusses a basic strategy for tackling this problem. It pays special attention to the aspect that modern software applications usually intensively make use of third-party components. Though the relations between these so-called "parts" are manifold, the proposed "software information leaflet" approach might give an idea on how operational support tools could be designed strategically. Finally, the paper lists open research questions addressing the aforementioned issues. Aspects that still have to be examined relate to knowledge modeling, the development of appropriate information encodings, as well as pragmatic utilization aspects.**

***Index Terms*—Sustainable software, sustainable development, software parts, software information leaflets.**

I. Introduction

Computers offer us many things we do not want to miss. Often, they support human activities that have positive effects on our environment and society (“software engineering for the planet” [22]). Nonetheless, in many aspects information and communication technology (ICT) affects the environment and society in a negative way, e.g. by consuming energy and natural resources, generating electronic waste, or causing social inequality.

Usually, computers need software in order to operate (see fig. 1). Thus software has an impact on aspects that are relevant with regard to sustainability issues. Consequently, for software producers a sustainable product design should be a basic requirement [17]. But these days, for software developers the design and implementation of green and sustainable software is a rather difficult task.

Practical guidelines and hints to assist software architects and programmers in their daily business to set up more sustainable systems unfortunately are still rare today. One reason might be that knowledge about the used software artifacts and design patterns and their impact on sustainability-relevant aspects such as energy consumption, hardware requirements (which might drive functional obsolescence effects with respect to the hardware), or social factors often is not available. Moreover, since sustainability information about software seems to be hard to find, platforms to gather and discuss this knowledge as well as mechanisms that make this information accessible inside the software engineering process are desirable.

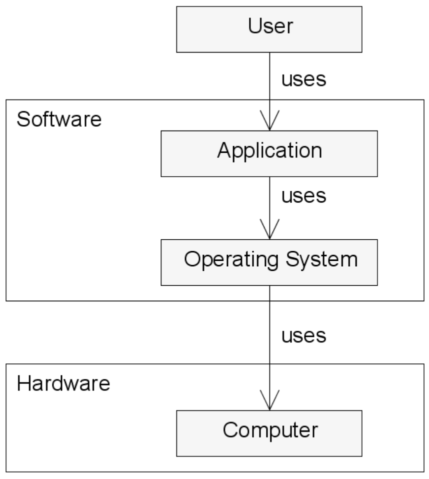

Fig. 1. Relation of user, software, and hardware

In this contribution a basic strategy for tackling these problems will be sketched. Special attention will be paid to the aspect that modern software applications usually intensively make use of external components. The rudimentary approach presented in the present paper illustrates how operational support tools could be designed strategically. Nonetheless, the list of open research issues concerning this topic still comprises many bullets, as we will see.

The paper is organized as follows. At the beginning, a definition of sustainable software is given (section 2). Then the notion of part relations is introduced (section 3). Based on the idea of patient information leaflets, the “software information leaflet” approach to provide information about software parts is described (section 4). Subsequently, general problems of this approach are discussed (section 5) and open research questions are listed (section 6). Finally, first ideas for a concrete implementation are given (section 7).

II. Sustainable Software Engineering

A. Sustainable Development

Sustainable development has been defined in many ways. One of the most frequently quoted definitions is from the Brundtland Commission's report: “[Sustainable development] meets the needs of the present without compromising the ability of future generations to meet their own needs” [24].

In the context of the research projects which are currently carried out at the Laboratory for Sustainable Development (LaNE) at Bochum University of Applied Sciences, we use the definition of Integrative Sustainability. Consequently, we intend to overcome the classical concept of three pillars of sustainability, where economy seems to have the same value as ecology and society. The qualities listed below characterize the concept of Integrative Sustainability [18, 19]:

- *Consistency, efficiency,* and *sufficiency* strategy should all be included.
- Unity of mankind, society, and nature and the idea of a *fully functioning society* will be assumed.
- The creation and conservation of *social fairness* (concentrating on the needs-centered approach) will be regarded as crucial aspects of sustainable development.

Notably, the term “sustainable” is not a synonym for “long-lasting” or “efficient” here.

B. Definition of Sustainable Software

Often, sustainable software is defined as follows [7, 12]: “*Sustainable Software* is software, whose direct and indirect negative impacts on economy, society, human beings, and environment that result from development, deployment, and usage of the software are minimal and/or which has a positive effect on sustainable development.”

With the aforementioned idea of Integrative Sustainability in mind, the conservation of nature and social fairness has to be emphasized in this definition. Thus, amongst others, important characteristics of software products should be:

- Minimization of irreversible and irretrievable commitment of energy and material resources (efficiency aspect);
- Positive effects due to its power to decrease material impacts of production, transport and consumption processes (dematerialization, demobilization, virtualization);
- Ensuring access to the software artifacts and corresponding knowledge for broad target groups (accessibility, social fairness);
- Avoidance of hardware obsolescence.

The first aspect refers to a so-called first order or “primary” effect as defined by Berkhout and Hertin [1]. The following two aspects relate to second and third order effects of ICT, i.e. indirect and “systemic” effects [9]. The classification of the last aspect is difficult since the reasons for obsolescence might be different in nature (technological or functional reasons, style obsolescence/desirability etc.).

In an alternative, naive view, the process of software production and consumption can be understood as a transformation of (individual) human knowledge and data (which, according to Von-Neumann's concept, include software-artifacts) into new “knowledge goods” and further data (including software) [18]. This view can be constituted by the models which have been developed in the context of semiotic research, where software is understood as a sign transformation process that operates on semiotic entities, as well as the assignment of meaning and knowledge [14]. This socio-technical transformation process is associated with an interchange of human beings, society, and natural environment. It requires the supply of material (including energetic) resources which also will be transformed during software development and operation [10].

Noteworthy, the regenerative character of the immaterial resource “software” which is highlighted by supporters of the “digital sustainability” approach [3], may not conceal the necessity to dematerialize and decarbonize software engineering processes.

However, both the transformational and the digital sustainability view augment the definition of sustainable software above [18].

III. Software Applications and Software Parts

A. Parts-and-Assemblies Approach

Presently, for economic reasons software development follows the “parts-and-assemblies” approach which means that software applications are built from pre-built standard parts [15]. Examples for such software parts are class and method libraries, operation system specific functions, database access components, or external Web services, just to mention a few. Often such artifacts are provided as third party software, i.e. software that is not associated with the software manufacturer or user (respectively customer).

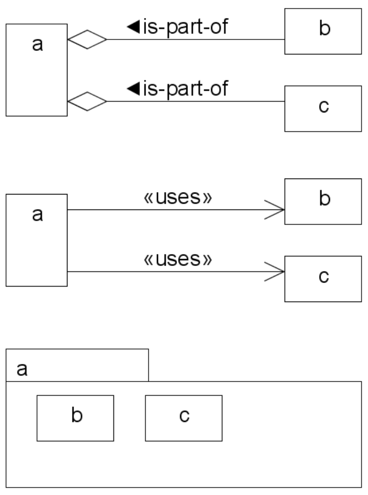

Fig. 2. Examples for different views on part relationships

Subsequently, independently from the underlying technological concept or functional granularity, we will denote such software artifacts which are seen as autonomous units as *parts.*

Note that for software engineers, whole-part or composition relationships often have a different meaning than for software operators or end users who might have a system construction from smaller units in mind; see figure 2.

In the first two UML diagrams [2], *a*, *b*, *c* are separate interacting units. In the third diagram a hierarchical whole-part relation is modeled, while the packaging aspect is made explicit.

However, in the scope of this paper the term “software part” primarily is used with respect to issues that are relevant to sustainability. Remarkably, the installation of some software *a* implies that the “parts” *b* and *c* will be used, too, and that the impacts of *b* and *c* also have to be considered.

It may be noted that in the system design phase software engineers usually put a strong focus on structural system views, particularly the so-called “software architecture” or “application architecture” [21]. This view seems to be quite compatible with the view presented here.

Anyway, regardless whether such standard parts are available as open or closed source, deeper application knowledge as well as information about sustainability-relevant characteristics is required. This topic will be picked up again later.

B. Formal Description of Part Relations

Using the language of mathematics, part relations can be described more concisely. Let S be the set of software artifacts, i.e. components, libraries consisting of classes and methods, Web services or other artifacts used to build and/or run software. Then, we can define $\rightarrow$, spoken “directly uses”, as a relation on S which is neither transitive nor reflexive. For $a, b \in S$, $a \rightarrow b$ means that b is a part of a.

Moreover, we can introduce the dependency relation $\rightarrow\!\!\rightarrow$ by defining $a \rightarrow\!\!\rightarrow b$ if 1. $a \rightarrow b$, or 2. there exists a chain of n elements c_i, $0 < i \leq n$, with $a \rightarrow c_1$, $c_1 \rightarrow c_2$, ..., $c_{n-1} \rightarrow c_n$, $c_n \rightarrow b$. One can easily show that $\rightarrow\!\!\rightarrow$ is a transitive relation, i.e. $(a \rightarrow\!\!\rightarrow b) \wedge (b \rightarrow\!\!\rightarrow c)$ implies $a \rightarrow\!\!\rightarrow c$.

This leads to a directed graph structure that is not necessarily acyclic. E.g., for a peer-to-peer communication between p_1 and p_2, both $p_1 \rightarrow p_2$ and $p_2 \rightarrow p_1$ will hold. Nonetheless, in practice often cycle-free tree-like structures will be given. (Note that for acyclic structures the corresponding undirected graph might contain cycles, e.g. if $a \rightarrow b \rightarrow d$ and $a \rightarrow c \rightarrow d$.)

C. Unneeded Parts

In practice, arbitrary utilization of application development environments, frameworks, or external libraries which add code to applications without need, often seems to be adverse with respect to energy efficiency and processor load [4, 13]. Consequentially, unneeded parts should be eliminated from the software.

It should be noted that after software removals often software parts which are no longer needed remain on the computer. This should also be avoided.

D. Design Alternatives

Concerning the part dependencies, it will be useful to explicitly describe situations where the software developer has the freedom to choose between parts. Simply spoken, if no alternatives are available, the developer cannot choose the better one.

For example, ways to select an implementation from a variety of alternatives could be modifications of the application's source code or configuration file edits. Often it is advantageous to offer external configuration options to the user, whereat hints to support configuration decisions should be given.

If either $a \rightarrow b$ or $a \rightarrow c$ holds (“exclusive or”), an operator | might be defined by introducing $a \rightarrow (b \mid c)$ as equivalent notation. Hereby, another node type would be introduced into the graph structure described above.

Example: Figure 3 shows the dependency graph for the set of relations $\{a \rightarrow (b \mid c),\ b \rightarrow d,\ c \rightarrow d,\ c \rightarrow (e \mid f),\ e \rightarrow g\}$. The graphic representation resembles those feature diagrams which are used in the context of generative software development approaches or domain engineering [5].

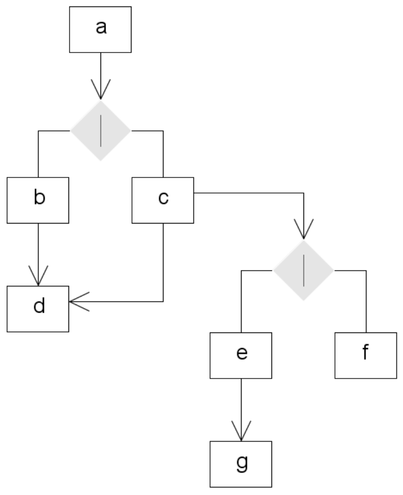

Fig. 3. Dependency graph example

In this context, usually the tree structure's root node corresponds to the software application which is the object of investigation. Pursuing the dependencies until their end, the most far flung leaf nodes would refer to the underlying computer hardware. In other words, with a view to figure 1, the hardware layer would terminate the directed graph structure.

Now a fundamental question is where the graph structure and thus the sustainability study should end (scoping problem as described in section 5). Another key issue is the identification of significant sustainability-relevant graph nodes that lie in the path from the root node to the leave nodes.

E. Design for Replaceability

Software artifacts which are “designed for replaceability” could be valuable with regard to sustainability. Thus the software should continuously be checked for replaceability to avoid subsequent migration and modernization efforts [16].

Another considerable aspect is that interfaces should not be designed in such a complex manner that it will be laborious to make use of them or that broad user groups are segregated (see [11] for an example).

Noteworthy, interfaces are an efficient mechanism to introduce part replaceability into software designs and implementations. Here, long life cycles, versioning, openness, and participation options seem to be good practice [18].

Mahmoud and Ahmad [13] propose a four-step design process that takes the mentioned considerations into account, see figure 4. After candidate parts have been found and the most appropriate parts have been selected, the selected parts are customized to meet the application requirements (including aspects relevant to sustainability) with a preferably high degree. The validation stage finishes the process.

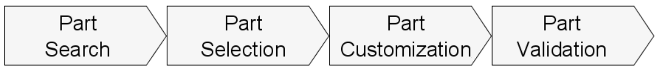

Fig. 4. Part-based software design process

IV. The PIL Idea

To describe software parts and their characteristics with respect to sustainability issues, one idea might be to provide additional information as it is done for pharmaceuticals.

Usually, medicinal products for human use come to the customer with a Patient Information Leaflet, in the following "PIL" for short (on German colloquially: Beipackzettel), also known as Package Leaflet. Directive 2001/83/EC requires that the PIL reflects "the results of consultations with target patient groups ['user consultation'] to ensure that it is legible, clear and easy to use" and that "the results of assessments carried out in cooperation with target patient groups shall also be provided to the competent authority." The PIL "must be written and designed to be clear and understandable, enabling the users to act appropriately [...]" [8].

These ideas apply not only to medical products and human users, but to software parts and our world as well. A "software information leaflet", hereafter referred to as SIL, could enable software developers to act appropriately when choosing software design alternatives or configuration parameters. SILs potentially could support software developers to design better, sustainable (or at least less unsustainable) systems.

A common problem with PILs is that many people just ignore them and do not make use of the given information. It requires further research, how people can be motivated to consult SILs.

However, machine processable SILs and suitable engineering tools could be a promising option to assist software developers and users; see section 6.C.

V. Problems With Part Descriptions

Some simple examples give an impression about obvious methodical problems that will take place when describing parts. (Depending on individual working experience, other examples will come to one's mind.)

Example 1: Assume a mapping application a which uses a Web service s_1, e.g. a standardized Web Map Service [6], to access up-to-date map images. Will this service (which is "just" delivering PNG or JPEG images on request) be part of the mapping application? I.e., is s_1 considered inside the software system model or will the relation $a \rightarrow s_1$ be neglected?

Example 2: To sort a rather huge array of strings which is partially sorted, you have to choose between two algorithms. Implementation a_1 requires $O(n \log n)$ operations, both for the best and the worst-case. Implementation a_2 runs with $O(n^2)$ at worst-case and $O(n)$ at best-case. Which implementation should be taken $(a_1 \mid a_2)$? And how can I decide for an implementation when I do not entirely know the operation conditions?

Example 3: A Web application with high request rate is programmed against the Servlet interface. Thus, this application could be installed in different Servlet containers (e.g., the Apache Tomcat or other engines), while the classes implementing the Servlet interface will be bound at run-time. How can I evaluate the application's characteristics when the underlying implementation might vary or is even unknown? (And again we've got the choosing problem here.)

Example 4: The Web Map Service in example 1 is given as an HTTP URL that now can be replaced by giving another URL for a service s_2 at run-time ($a \rightarrow s_2$ instead of $a \rightarrow s_1$). Does the operator of the application a have the expertise to change the service URL? Noteworthy, the service might be replaced dynamically (i.e. during system operation) here, so in addition we've got the dynamic binding problem again.

Example 5: In example 3, the servlet could return an HTML page (possibly with JavaScript code embedded) to the client. Has HTML to be considered as a part of the application? (And what is about the JavaScript portions that are steadily increasing these days?) Here we've got the scoping problem again.

Hence, some typical problems have to be expected when describing parts:

- *Scoping problem:* How can the boundaries of software parts be determined? (Cf. discussion at the end of section 3.D.)
- *Comparability of descriptions:* How to ensure that descriptions of different parts are comparable? This might refer to structural elements, nomenclatures for properties, reference systems chosen for qualitative or quantitative values etc.
- *Selection criteria determination:* If parts can be used alternatively, how can we give criteria to decide which part to take ("choosing problem")?
- *Consideration of dynamic bindings:* How can I analyze an application's behavior when the used part implementations might change at run-time?
- *Description of situation dependencies:* How can I describe the operation conditions of software parts which might depend on concrete use cases, other active processes etc. and which influence a part's behavior? (It is as with the 10 mm dowel pin that might be suitable for a certain situation and improper for another one.)
- *Granularity problem:* Parts might refer to different abstraction levels, e.g. varying from complex frameworks, services, or components to class libraries, classes, and single methods.

Another methodical problem springs from the fact that most recommendations given by SILs will not be absolute and rather deserve discussion. For example, probably the description of situation dependencies or the assessment of social segregation aspects could be better discussed by an open Web community than by a closed institution.

VI. Open Research Questions

Taking the problems listed in the previous section into account, for SIL-like approaches there are various open research questions that have to be answered:

A. Semantic Level (Content Model)

- Which content has to be provided for the parts with respect to sustainability-relevant issues?
- How is this content structured? (One could pick up the simple graph structure presented in section 3, where various structural concepts would superimpose the graph, e.g. to describe operation conditions, different dependency types, to address granularity issues, etc.)
- How can we keep the content given in different SILs comparable?
- Which kinds of part dependencies have to be modeled?

B. Syntactic Level (Code Level)

- How to encode the content model in machine processable form?
- Since part dependencies might be hidden and difficult to detect, how can we set up tools for automatic dependency detection ("dependency mining")?

C. Pragmatic Level (Application and Action)

- How can the encoded knowledge support the work of software developers and administrators?
- How can we motivate people to contribute content to the knowledge repositories (which would evolve dynamically over time)? How can a broad participation be achieved?
- How could specific software engineering tools encourage software developers to act? (Or more general [20]: How can we turn technical construction knowledge into development action?)
- How could SILs be integrated into operational software engineering tools such as integrated development environments (IDEs), profiling tools, loggers, build tools etc.?
- How could SIL-based software metrics to characterize the degree of sustainability of software artifacts be designed?
- What could a concept for a (Web based) community that collects and discusses SIL knowledge look like? Which tools should such a platform offer?
- How can we involve people in knowledge discovery processes in the context of green and sustainable software? How could concepts from the field of "Citizen Science" be introduced into such a community?

VII. First implementation Considerations

A. External SIL Provision

As noted earlier, the question which concrete informational details the software developer needs about the software artifacts he/she plans to use, cannot easily be answered. Here are just some examples which illustrate the subject: How should usage conditions influencing energy consumption be described? Which software characteristics are relevant with respect to social aspects such as fair software access etc.? What is the best way to provide details about artifacts which have been designed as replaceable parts and how to determine, when to replace and by which artifact?

Anyway, it would be useful to give this information in a machine processable form, e.g. a format based on the eXtensible Markup Language (XML). This way, software development tools such as IDEs or CASE tools could import the information. Then a fundamental task would be the development, specification, and maintainment of a suitable language, e.g. an XML-based modeling language (usually given as an XML schema definition). Ideally, this language would act as a "lingua franca" for all stakeholders (i.e., software developers, software administrators, chief information officers, potential customers, end users, project managers etc.).

Since external artifacts should be used as provided, i.e. they should be left untouched, it will be difficult to let standardized additional information units become part of those artifacts. If, for example, an external class library without an SIL is used, probably the best way would be to manage the SIL externally and add a reference to the library described by this SIL; see figure 5 (UML diagram). Another advantage of this approach is that the SIL could be reviewed and revised externally, e.g. by an open, neutral and sustainability aware Web community.

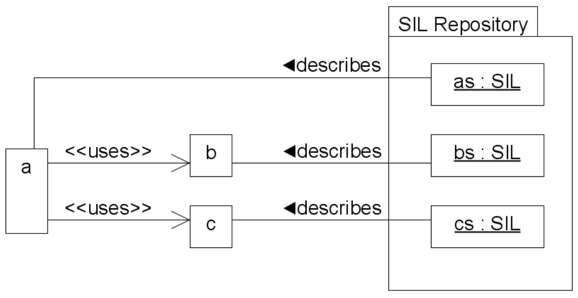

Fig. 5. SIL repository

B. Dependency Detection

A basic task is the identification of dependencies between parts. Since dependencies might be hidden, tool support would be desirable.

Both source code file analyzers and more sophisticated tools such as loggers, code coverage analyzers, or profilers could serve as starting points for further considerations with respect to dynamic aspects. In this context, transitivity of the manifold dependency relation $\rightarrow\rightarrow$ (see section 3.B) has to be considered.

VIII. Conclusions and Future Work

When realizing applications, software producers usually intensively make use of external software parts.

In practice, the usage relations ($\rightarrow$) are manifold: “Parts”, as they have been defined in this paper, might be integral components of software applications, or they just require other components to be installed on the target computer. On the other side of the spectrum, parts are loosely coupled autonomous units, e.g. a remote service which is dynamically called by an application. It will be necessary to distinguish between different types of $\rightarrow$ relations.

The design and implementation of sustainable software requires deeper knowledge about the used parts. One approach to provide this knowledge would be the introduction of machine processable “software information leaflets”, i.e. “PILs for software parts” (German “Beipackzettel”).

Though this approach sounds simple, it leads to various questions that have to be answered before realization activities could start, e.g. concerning part scoping, description of dynamic and usage-dependent software behavior or the description of “soft” factors such as fair software access or social segregation effects. Here, a suitable content model is required.

Concerning the question which content has to be provided for the parts, research activities that are currently carried out in the context of software product labeling [12], e.g. the creation of a kind of “eco-label”, seem to be very useful. Currently the German Federal Environment Agency is compiling concrete criteria for green and sustainable software that could serve as a starting point [23]. Amongst others, they address the energy efficiency of applications, hardware performance requirements, hardware obsolescence effects, or self-determined software use (“user empowerment”).

Yet there are more challenging tasks: How can the required sustainability knowledge be acquired? How should tools that fit into modern software engineering processes be designed? How can stakeholders be motivated to use this knowledge?

And finally, irrespective of whether we will have SILs or not: How can we turn technical knowledge about sustainable software into development action?

Acknowledgment

The project “Forschungsschwerpunkt Nachhaltige Entwicklung” which is funded by the Ministry of Innovation, Science and Research of the state North Rhine Westphalia (MIWF NRW) within the “FH Struktur” program gave rise to this text.

In particular, the author thanks Petra Schweizer-Ries, Christian Danowski, and Oliver Stengel for their support during the past years.

And finally, this paper would not have been possible without Nina and Werner Windisch.

References

[1] F. Berkhout and J. Hertin, “Impacts of information and communication technologies on environmental sustainability: speculations and evidence,” Report to the OECD, 2001.

[2] G. Booch, J. Rumbaugh, and I. Jacobsen, “The Unified Modeling Language user guide,” 2nd ed., Addison Wesley, 2005.

[3] T. Busch, “Open Source und Nachhaltigkeit,” Open Source Jahrbuch, Berlin: Technische Universität, 2008.

[4] E. Capra, C. Francalanci, and S. A. Slaughter, “Is software "green"? Application development environments and energy efficiency in open source applications”, Information and Software Technology, vol. 54, no. 1, 2012, pp. 60–71.

[5] K. Czarnecki and U. Eisenecker, “Generative programming: Methods, tools, and applications”, Boston, MA: Addison-Wesley, 2000.

[6] J. de la Beaujardiere, Ed., “OpenGIS Web Map Service (WMS) implementation specification”, Open Geospatial Consortium, doc. no. 06-042, 2006, http://www.opengeospatial.org/docs/is

[7] M. Dick, J. Drangmeister, E. Kern, and S. Naumann, “Green software engineering with agile methods,” 2nd International Workshop on Green and Sustainable Software (GREENS 2013), San Francisco, CA, 2013, pp. 78–85.

[8] European Commission, “Guideline on the readability of the labelling and package leaflet of medicinal products for human use,” directive 2001/83/EC, articles 59(3), 61(1), 63(2), Brussels, 2009.

[9] L. Hilty, “Information and communication technologies for a more sustainable world,” in Information Resources Management Association, “Green technologies: Concepts, methodologies, tools and applications,” Vol. I (chapter 1.4), Hershey, PA: Information Science Reference, 2011, pp. 36-45.

[10] L. Hilty, W. Lohmann, S. Behrendt, M. Evers-Wölk, K. Fichter, and R. Hintemann, “Green software : Analysis of potentials for optimizing software development for resource conservation,” report no. (UBA-FB) 001883/2,E, Berlin, Germany: Federal Environment Agency, 2015.

[11] T. Johann and W. Maleej, “Position paper: The social dimension of sustainability in requirements engineering,” 2nd International Workshop on Requirements Engineering for Sustainable Systems, Rio, Brasil, July 2013.

[12] E. Kern, M. Dick, S. Naumann, and A. Filler, “Labelling sustainable software products and Websites: Ideas, approaches, and challenges,” 29th International Conference on Informatics for Environmental Protection (EnviroInfo 2015) and 3rd International Conference on ICT for Sustainability (ICT4S 2015), pp. 82–91.

[13] S. S. Mahmoud and I. Ahmad, “A green model for sustainable software engineering”, International Journal of Software Engineering and Its Applications, vol. 7, no. 4, July 2013, pp. 55–74.

[14] M. Nadin, “Semiotic machine,” Public Journal of Semiotics, I(1), Jan. 2007, pp.57–75.

[15] J. M. Neighbors, “Software construction using components,” dissertation, Irvine, CA: University of California, Department of Information and Computer Science, 1980.

[16] F. Pientka, “Gebaut für den Wandel : Nachhaltige Software-Entwicklung,” Business Technology (Magazin für IT-Leadership & Innovation), Heft 18, 3.2014, Frankfurt/M., Germany: Software & Support Media, pp. 35–38.

[17] A. Raturi, B. Penzenstadler, B. Tomlinson, and D. Richardson, “Developing a sustainability non-functional requirements-framework,” 3rd International Workshop on Green and Sustainable Software (GREENS 2014), pp. 1–8.

[18] B. Schmidt and A. Wytzisk, “Software Engineering und Integrative Nachhaltigkeit,” 2nd Workshop “Environmental Informatics between Sustainability and Change” (UINW 2014), proceedings INFORMATIK 2014, Lecture Notes in Informatics (LNI), Vol. P-232, Stuttgart, Germany, Sept. 2014, pp. 1935–1945.

[19] P. Schweizer-Ries, “Sustainability science and its contribution to IAPS: Seeking for Integrated Sustainability,” IAPS Bulletin, 40, Autumn 2013, pp. 9–12.

[20] P. Schweizer-Ries and D. P. Perkins, “Sustainability science: Transdisciplinarity, transepistemology, and action research,” Umweltpsychologie, 16(1), 2012, pp. 6–10.

[21] I. Sommerville, “Software Engineering,” 10th ed., Harlow, UK: Pearson Education, 2016.

[22] J. Taina, “Good, bad, and beautiful software : In search of green software quality factors,” CEPIS Upgrade, XIII(4), Oct. 2011, pp. 22–27.

[23] Umweltbundesamt, “Sustainable Software Design : Entwicklung einer Methodik zur Bewertung der Ressourcen-Effizienz von Softwareprodukten,” information leaflet, Berlin, Germany: Federal Environment Agency, Nov. 2015. http://www.uba.de/publikationen/sustainable-software-design

[24] World Commission on Environment, Eds., “Report of the World commission on environment and development: Our common future,” Oslo: UN, 1987.