# Software as a Social Artifact: a Management and Evolution Perspective

Xiaowei Wang[1], Nicola Guarino[2], Giancarlo Guizzardi[3], and John Mylopoulos[1]

[1] Department of Information Engineering and Computer Science, University of Trento, Italy
{xwang,jm}@disi.unitn.it
[2] ISTC-CNR, Trento, Italy
guarino@loa.istc.cnr.it
[3] Ontology and Conceptual Modeling Research Group (NEMO),
Federal University of Espírito Santo (UFES), Brazil
gguizzardi@inf.ufes.br

**Abstract.** For many, software is just code, something intangible best defined in contrast with hardware, but it is not particularly illuminating. Microsoft Word turned 30 last year. During its lifetime it has been the subject of numerous changes, as its requirements, code and documentation have continuously evolved. Still a community of users recognizes it as "the same software product", a persistent object undergoing several changes through a social process involving owners, developers, salespeople and users, and it is still producing recognizable effects that meet the same core requirements. It is this process that makes software something different than just a piece of code, and justifies its intrinsic nature as a social artifact. Building on Jackson's and Zave's seminal work on foundations of requirements engineering, we propose in this paper an ontology of software and related notions that accounts for such intuitions, and adopt it in software configuration management to provide a better understanding and control of software changes.

**Keywords:** software, software evolution, software configuration management, software versioning, artifact, ontology, software requirements.

## 1 Introduction

Software has become an indispensable element of our culture, as it continues its relentless invasion of every facet of personal, business and social life. Despite this, building software applications is still an art more than science or engineering. Failure and partial failure rates for software projects still are stubbornly high and Software Engineering (SE) practice is often ahead of SE research, in virgin territory, where practitioners have few engineering principles, tools and techniques to turn to.

We believe that one of the reasons for this unhappy situation is a lack of consensus on what exactly is software, what are its defining traits, its fundamental properties and constituent concepts and how do these relate to each other. For many, both within and without the SE community, software is just code, something intangible best defined as

the other side of hardware. For example, the Oxford English Dictionary defines software as "the programs and other information used by a computer" and other dictionaries adopt paraphrases. In a similar spirit, software maintenance tools such as Concurrent Versions System (CVS) and Apache Subversion (SVN), the version control systems of choice for almost 30 years, are used primarily for code management and evolution, while requirements, architectural specifications etc. are left out in the cold.

Unfortunately, treating software as simply code is not very illuminating. Microsoft (MS) Word turned 30 last year (2013). During its lifetime it has seen numerous changes, as its requirements, code and documentation have continuously evolved. If software is just code, then MS Word of today is not the same software as the original MS Word of 1983. But this defies the common sense that views software as a persistent object intended to produce effects in the real world, which evolves through complex social processes involving owners, developers, salespeople and users, having to deal with multiple revisions, different variants and customizations, and different maintenance policies. Indeed, software management systems were exactly intended to support such complex processes, but most of them consider software just as code, dealing with software versioning in a way not much different than ordinary documents: the criteria underlying the versioning scheme are largely heuristic, and the *change rationale* remains obscure.

Yet, differently from ordinary documents, software changes are deeply bound to the nature of the whole software development process, which includes both a requirements engineering phase and subsequent design and implementation phases. This means that, making a change to a software system may be motivated by the need to fix a bug (code), to adopt a more efficient algorithm or improve its functionality (program specification), adapt it to a new regulation (requirements) and so on. As we shall see, each of these changes affects a different *artifact* created within the software development process. In this paper we shall present an ontology that describes what these different artifacts are, and how they are inter-related.

The main contribution of this work consists of an argument, supported by ontological analysis, that software has a complex artifactual nature, as many artifacts result from a design process, each having an intended purpose that characterizes its identity. This is what distinguishes software artifacts from arbitrary code: they are recognizable as having a purpose, they are the result of an intentional act. A further characteristic of software is its social nature. In order to exist, software[1] presupposes the existence of a community of individuals who recognize its intended purpose. The members of such community may change in time, and, as already noted, may include developers, users, salespeople and stakeholders. In addition, certain software artifacts (*licensed software products*) have a further social character: they presuppose a pattern of mutual *commitments* between owners and users.

The rest of this paper is organized as follows: Firstly, we provide an ontological analysis of a number of concepts related to software and software engineering, under-

[1] In our analysis here, we eschew the limit case of software that is privately produced and used.

lining their artifactual and social nature in Section 2 and 3. The result of such analysis is a layered ontology of software artifacts, presented in Section 4. Section 5 discusses the practical impact of our proposed ontology on software management and software modeling. Section 6 summarizes our contributions and sketches future work.

## 2 The Artifactual Nature of Software

### 2.1 State of the Art: Approaches to the Ontology of Software

In the literature, the term "software" is sometimes understood in a very general sense, independently of computers. For example, Osterweil [1] proposes that, in addition to computer software, there are other kinds of software, such as laws or recipes. Focusing on computational aspects, several scholars (e.g. Eden and Turner [2], Oberle [3]) have addressed the complex relationships among i) software *code*, consisting of a set of computer instructions; ii) a software *copy*, which is the embodiment of a set of instructions through a hard medium; iii) a *medium*, the hardware on which a software copy runs; iv) a *process*, which is the result of executing the software copy.

A different approach to account for the artifactual nature of software is taken by Irmak [4]. According to Irmak, software is synonymous to program and can be understood in terms of the concepts of algorithm, code, copy and process, but none of these notions can be identified with software, because due to its artifactual nature, software has different identity criteria than these concepts. Therefore, a program is different from a code. We share many of Irmak's intuitions, as well as the methodology he adopts to motivate his conclusions, based on an analysis of the condition under which software maintains its identity despite change. However, he leaves the question of "what is the identity of software" open, and we answer this question here.

### 2.2 Code and Programs

Consider a *computer code base*, defined as a well-formed sequence of instructions in a Turing-complete language [2]. Two bases are identical iff they consist of exactly the same sequences of instructions. Accordingly, any syntactic change in a code base *c1* results in a different code base *c2*. These changes may include variable renaming, order changes in declarative definitions, inclusion and deletion of comments, etc.

A code *implements* an algorithm. Following Irmak [4], we treat an algorithm as a language-independent *pattern of instructions*, i.e. an abstract entity correlated to a class of possible executions. So, two different code bases *c1* and *c2* are *semantically equivalent* if they implement the same algorithm. For instance, if *c2* is produced from *c1* by variable renaming, *c2* will be semantically equivalent to *c1*, and still possess a number of properties (e.g., in terms of understandability, maintainability) that are lacking in *c1*.

Some authors, e.g. Lando et al. [5], who identify the notion of program with that of computer code, while others, such as Eden [2] and Oberle [3] distinguish program-script (program code) from program-process (whose abstraction is an algorithm). However, we agree with Irmak that we cannot identify a program neither with a code,

a process, or an algorithm. The reason is that such identification conflicts with common sense, since the same program may have different code bases at different times, as a result of updates[2]. What these different code bases have in common is that, at a certain time, they are selected as *constituents* of a program that is intended to implement the very same algorithm.

To account for this intuition, we need a notion of (technical) artifact. Among alternatives in the literature works, Baker's proposal [6] works best for us: *"Artifacts are objects intentionally made to serve a given purpose"; "Artifacts have proper functions they are (intentionally) designed and produced to perform (whether they perform their proper functions or not)"; "What distinguishes artifactual [kinds] from other [kinds] is that an artifactual [kind] entails a proper function, where a proper function is a purpose or use intended by a producer. Thus, an artifact has its proper function essentially"*. These passages are illuminating in several respects. Firstly, Baker makes clear that artifacts are the results of intentional processes. Moreover, she connects the identity of an artifact to its proper function, i.e., one that fulfills its intended purpose. Finally, she recognizes that the relation between an artifact and its proper function exists even if the artifact does not perform its proper function. In other words, the connection is established by means of an intentional act.

In light of these observations, code is not necessarily an artifact. If we accidentally delete a line of code, the result might still be a computer code. It will not, however, be "intentionally made to serve a given purpose". In contrast, a program is *necessarily* an artifact, since it is created with a particular *purpose*. What kind of purpose? Well, of course the *ultimate* purpose of a program is –typically– that of producing useful effects for the prospective users of a computer system or a computer-driven machine, but there is an *immediate* purpose which belongs to the very essence of a program: producing a certain result *through execution on* a computer, *in a particular way*. We insist on the fact that the desired result and the relative behavior must come about *through* a computer, as they concern desired phenomena arising within the memory segment allocated to the program while the program runs. As usual, an abstract description of such phenomena is given by specifying a data structure and an algorithm that manipulates it [7]. Note that an algorithm, in turn, is defined as a procedure that implements a certain *function*, intended to bring about a desired change within the data structure. In summary, the immediate purpose of a program is described by a data structure, a desired change within such data structure, and a procedure to produce such change by manipulating the data structure. Altogether, such information is called a *program specification*. In contrast with code, every program has, necessarily, a purpose: satisfying its specification, namely implementing the desired function in the desired way. In order for a program to exist, its specification must exist, even if only in the programmer's mind.

According to the discussion above, we have to conclude that a program is not identical to code. This begs the question: what is the relation between the two, then? In

---

[2] Irmak also admits that the same program may have different algorithms at different times, but we shall exclude this, distinguishing a program from a software system (see below).

general, the relation between an artifact and its material substratum is one of *constitution*. As put by [6], the basic idea of constitution is that whenever a certain aggregate of things of a given kind is in certain circumstances, a new entity of a different kind comes into being. So, when code is in the circumstances that somebody intends to produce certain effects on a computer, then a new entity emerges, constituted by the code: a *computer program.* If the code does not actually produce such effects, it is the program that is faulty, not the code. In conclusion, a program is *constituted* by code, but it is not identical to code. Code can be changed without altering the identity of its program, which is anchored to the program's essential property: its intended specification.

### 2.3 Programs and Software Systems

We have seen that, since the identity of a program depends on its intended specification, and the specification includes both the desired function and the algorithm through which such function is supposed to be implemented, we cannot change the algorithm without changing the program, even if the function is the same. Yet, in the course of software development, it is often the case that software keeps its identity after a change in the algorithm: typically we say tha*t the software* is now more efficient after such a change. The strategy we shall adopt to account for such phenomena will be the same as before: we add a new entity to our layered ontology, a *software system*, which is constituted by a *software program*, which in turn is constituted by code. The essential property of a software system is being intended to satisfy a *functional specification (internal specification)*, concerning a desired change in a data structure inside a computer[3], abstracting away from the behavior. Note that, in the way we defined it, a program specification already includes a functional specification, so specifying a software system is just specifying a program in an abstract way, without constraining its behavior. This means that program specification and a software system specification overlap in the functional specification.

To give a concrete idea of our approach, let us introduce the example we shall use in the rest of the paper. Consider the following *functional specification (S)*, expressed here in natural language: *the system receives as input a connected, undirected graph G such that to each arc connecting vertices in G a positive numeric weight is assigned. The system returns a subgraph of G that is a tree and connects all vertices together. Moreover, the sum of weights in the returned tree must be equal or less than the sum of the weights in all possible trees of the same nature that are subgraphs of G*. This specification defines the desired function of finding a Minimum Spanning Tree (MST). Now, suppose that we start working on implementing this specification. First we decide the algorithm to implement, say Prim's Algorithm, and then we start writing the code. At a certain point, when this code sufficiently characterizes the program (i.e., we believe it may be correct, and it is ready to be tested), then, by an act of

[3] We exclude from this discussion any function concerning events in the outside world, such as a robotic arm moving an object from position A to B.

creation, we decide that this code now constitutes our program (let us call it *MST-Finder*). From that point on, we can keep changing the constituting code in order, for example, to fix bugs, improve readability and maintainability, etc. We can also improve its memory and time efficiency, while keeping the same algorithm. We can even change the programming language the initial and subsequent code bases are implemented in. Each of these changes creates a different code base but we still have the same program as long as we maintain the intention to implement the very same algorithm. On the contrary, if we replace the code by implementing Kruskal's instead of Prim's Algorithm, then what we get is a different program, although this new program, which however still constitutes the same software system.

### 2.4 Software Systems and Software Applications

As we have seen, programs and software systems, as defined, are software artifacts intended to produce effects inside a computer, i.e., changes concerning symbolic data structures, which reside in computer memory. Yet, as Eden and Turner observe [2], a peculiar aspect of software, with respect to other information artifacts such as books or pictures, is its *bridging role* between the abstract and the concrete: despite the fact that software has an abstract nature, it is designed to be *applied* to the real world. Therefore, it seems natural to us to take a requirements engineering perspective while analyzing the essence of software, instead of focusing on computational aspects only. So, we shall base our further analysis on a revisitation of the seminal works by Jackson and Zave (hereafter J&Z) on the foundations of requirements engineering [8], [9], [10] which clearly distinguishes the *external environment* (where the software *requirements* are typically defined), the *system-to-be* (a computer-driven machine intended to fulfill such requirements), and the interface between the two.

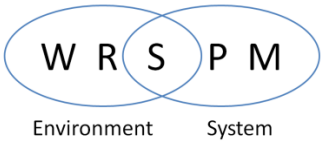


**Fig. 1.** A reference model for requirements and specifications (from [10]).

Figure 1 presents the J&Z's reference model [10].The model consists of two overlapping sets of phenomena: environment phenomena, usually happenings in the world, and system phenomena that happen inside the computer. Importantly, the two sets overlap. This means that some phenomena happen at the interface between the computer and the environment and are visible both from within and without the computer.

The letters mark different kinds of phenomena, world assumptions (*W*), requirements (*R*), *specification* that describes desired behavior at the interface with the environment (*S*), program specification (*P*) that determines desired machine behavior, and assumptions on the machine behavior (*M*). Specifically, such assumptions concern a

*programmable platform*[4] *properly connected with the external environment by means of I/O devices.*

If the environment and system interact in the desired way, then the following condition needs to be satisfied: if world assumptions holds, and specification phenomena occur, then the requirements are satisfied [10]. In a compact form, J&Z describe this condition as: $W \wedge S \vDash R$. We say in this case that *S satisfies R* under the assumptions *W*.

This view constitutes a reference model for *requirements* engineering, emphasizing the role of the specification of machine behavior at its interface with the environment. From a *software* engineering perspective, however, we are interested not in the machine as such, but in the *program* which drives it, and ultimately in the relationship between the program and its requirements. As observed in [10], such relationship is in turn the composition of two relationships: *If (i) S properly takes W into account in saying what is needed to obtain R, and (ii) P is an implementation of S for M, then (iii) P implements R as desired.*

To properly account for this picture, it is important to make explicit the relationship between a program and its internal computer environment, which is only implicitly accounted by J&Z's approach. So we propose a revised model described in Fig. 2.

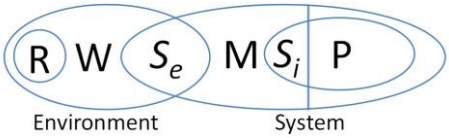


**Fig. 2.** Our revised reference model.

In Figure 2, the difference is that now the programmable platform is isolated as a proper part of the system-to-be, and its interface with the program is made explicit. Reflecting the standard computer architecture, we shall assume that such platform includes operating system and I/O device phenomena. So the platform has *two* interfaces: an *external* interface (whose specification describes phenomena in the external world, such as light being emitted by the monitor or keys being pressed), and an *internal* interface, whose specification describes phenomena within the program and the operating system. A software system specification ($S_i$) then just concerns this internal interface, while a program specification (*P*) also concerns phenomena that are not visible to the platform.

Now, let us go back to our software system intended to solve the MST problem. In order for it to accept input from the user and display the results, it has to generate a sequence of machine-based phenomena, using the functionality of its programming platform. In addition, of course we want our system to interact with the user in a proper way. Such expected behavior is described by the *external* specification $S_e$. In order for the program to behave properly, a condition very similar to the one de-

[4] J&Z use the term *programming* platform. We believe that *programmable* platform is more perspicuous.

scribed above must hold: $M \wedge S_i \vDash S_e$. This means that our MST program has to interact with the particular machine at hand (say, running a Windows operating system) to produce the desired I/O behavior.

Again, we can apply in this case the same line of reasoning which motivated the distinctions between code, program, and software system: when a software system is explicitly *intended* to implement an external specification for a certain machine, then a new software artifact emerges: we shall call it a *software application.* A software application is constituted by a software system intended to determine a specific external behavior for a specific machine. Such intention is an essential property of a software application, which distinguishes it from a software system. Note that we follow here the popular terminology according to which a software application "causes a computer to perform useful tasks beyond of the running of the computer itself" [11], but, for the reasons explained below, we restrict its range to the "useful tasks" concerning the external interface only, not the outside environment.

As a final note, consider that $S_e$ in the formula above plays the role of $R$ in the original J&Z's formula, $W \wedge S \vDash R$. This shows the power and the generality of J&Z's model. Depending on where we place stakeholder requirements, in the scheme of Fig. 2, we can apply this general model to express the relationship between the requirements and the specification of what we have to realize in order to satisfy them. This paper, for reasons of brevity, we assume that stakeholder requirements concern the external environment, as shown in Fig. 2. This is the standard case of so-called *application* software, as distinct from *system* software, whose requirements concern phenomena inside the computer itself.

### 2.5 Software Applications and Software Products

Finally, let us consider the role of stakeholder requirements in the framework we have described so far. Going back to our MST example, a plausible description of such requirements could be "*We want to minimize the amount of cable necessary to connect all our network routers*". So there is a desired state, obtained by manual intervention supported by computer assistance, such that the amount of cable used is the minimal. Obtaining this result by means of a certain software not only presupposes the solution of the abstract MST problem, but of course a lot of assumptions concerning the world and the people's skills and behaviors. Moreover, during the evolution of such software, assuming world and machine assumptions remain the same, different external specifications may be designed, corresponding to different user interfaces. In this case people may say that *the same software* is evolving. According to the methodology followed so far, this means that a new artifact emerges, constituted by a software application, which we shall call a *software product.* A software product is constituted by a software application intended to determine specific effects in the environment as a result of the machine behavior, under given world assumptions. Such intention is an essential property of a software product, which distinguishes it from a software application.

In conclusion, the notion of software product captures perhaps the most common use of the word "software" in the daily life. It is important to remark that a software

product is intended to achieve some effects in the external environment *by means of a given machine*, and *under given environment assumptions*. So, assuming they have exactly the same high-level requirements, MS Word for Mac and MS Word for PC are different software products, since they are intended for different machines. Similarly, country-oriented customizations of Word for Mac may be understood as different products, since they presuppose different language skills, unless the requirements already explicitly include the possibility to interact with the system in multiple different languages.

## 3 The Social Nature of Software

In addition to its artifactual nature, discussed in detail above, software –at least software used every day in our society– has also a strong social nature, which impacts on the way it is produced, sold, used and maintained. There are two main social aspects of software we shall consider under our evolution perspective: *social recognition* and *social commitment.*

### 3.1 Social Recognition and Software Identity

We have seen the key role the constitution relation plays in accounting for the artifactual nature of software. But how is this constitution relation represented and recognized? In the simplest of cases, we can think of a program produced by a single programmer for personal use. In this case, we can imagine that the constitution relationship binding a program with its constituting code exists solely in the mind of this programmer. Likewise, if this program comes to constitute a software system, then this constitution relation, again, exists only in the mind of the programmer. Yet, in order for a software artifact to exist in a social context, we shall assume that the constitution relation between the artifact at hand and its constituent needs to be explicitly communicated by the software author, and recognizable by a community of people. As a minimal situation, we consider these communications about constitution and intentions to satisfy specifications as true communicative acts that create expectations, beliefs and contribute to the creation of commitments, claims and a minimal social structure (possibly reflecting division of labor) between the software creator(s) and the potential users or stakeholders. Once this social structure exists, the creators' actions become social actions and are subject to social and legal norms that support expectations and rights. To cite one example, we use the motion picture "The Social Network" based on the book "Accidental Billionaires" [12] reporting on the creation of Facebook. As shown there, the legal battle involving the authorship rights in Facebook was at moments based on the discussion of shared authorship between M. Zuckerberg and E. Saverin regarding an initial program (Saverin was allegedly a prominent proposer of the algorithm) and software system, much before the product Facebook existed. At other times, the legal battle between Zuckerberg and the Winklevoss brothers was based on a shared system specification of another program even if, as argued by Zuckerberg, no lines of the original code had been used by Facebook.

In more disciplined software engineering settings, anyway, the constitution relationships and the intended specifications are documented by program headers and possibly user manuals or separate product documentation. Notice that, without the explicit documentation of these relationships, the software artifacts will depend on their creators in order to exist, since the constitution relationships are sustained by their intentional states. Once these relationships are documented, these artifacts can outlive their creators, as long as this documentation can be properly recognized and understood. So, for instance, although Joseph Weizenbaum is no longer alive, by looking to a copy of the ELIZA [13] code, one can still reconstruct the chain of intentions from the informal requirements specification all the way down to the code. In formal ontological terms, this means that software artifacts are just historically (but not constantly) depending on their authors, and in addition they are generically constantly depending on a community of people who recognize their essential properties. If such community of people ceases to exist, the artifact ceases to exist.

### 3.2 Social Commitment and Software Licensing

As we have seen, the different kinds of software artifacts we have discussed are based on a requirements engineering perspective. We cannot ignore however another perspective that deeply affects the current practice of software engineering, namely the *marketing perspective*. In the present software market, software products do not come alone, since what companies sell are not just software products: in the vast majority of cases, a purchase contract for a software product includes a number of rights and duties on both parties, including the right to download updates for a certain period of time, the prohibition to give copies away, the right to hold the clients' personal data and to automatically charge them for specific financial transactions, and so on. Indeed, the very same software product can be sold at different prices by different companies, under different *licensing policies*. The result is that software products come to the market in the form of *service offerings*, which concern *product-service bundles.* According to [14], a *service offering* is in turn based on the notion of *service*, which is a social commitment concerning in our case maintenance actions. Service offerings are therefore meta-commitments, i.e., they are commitments to engage in specific commitments (namely, the delivery of certain services) once a contract is signed. So, before the contract is signed we have another software entity emerging: a *Licensable Software Product.* After the contract is signed, we have a *Licensed Software Product*. Notice that the services regulated by the contract may not only concern the proper functioning of software (involving the right to updates), but also the availability of certain resources in the environment where the software is supposed to operate, such as remote servers (used, e.g., for Web searching, VOIP communication, cloud syncing...). So, when Skype Inc. releases Skype, it publicly commits to engage in such kind of commitments. By the way, this means that, when buying Skype from Skype Inc., Microsoft is not only buying the software product, but it is also buying all the rights Skype Inc. has regarding its clients.

Note that, even in absence of a purchasing contract, when releasing a product as licensable product, the owner creates already social commitments and expectations

towards a community of users and re-users of the product. For example, take the Protégé Ontology editor, which is a free open-source product released under the Mozilla Public License (MPL) [15]. This grants the members of the user community the right to change Protégé's code and to incorporate it even in commercial products.

## 4 A Layered Ontology of Software

The discussion so far induces a layered structure for our ontology of software artifacts, based on their different identity criteria and on the constitution relationship that links them to each other. Such layered structure is shown in Figure 3. As usual, the subsumption relation is represented by an open-headed arrow. The closed-headed arrows represent some of the basic relations discussed in the paper. Starting from code, several kinds of software artifacts have been proposed, all eventually constituted by code. The different essential properties characterizing their identity are shown to the right, linked by a relation of specific constant dependence. As the concepts have already been introduced, we give here only a brief account of the relations appearing in the picture. For some of them (constitution and specific constant dependence), the intended semantics is rather standard, while for others we just sketch their intended meaning, postponing a formal characterization to a future paper.

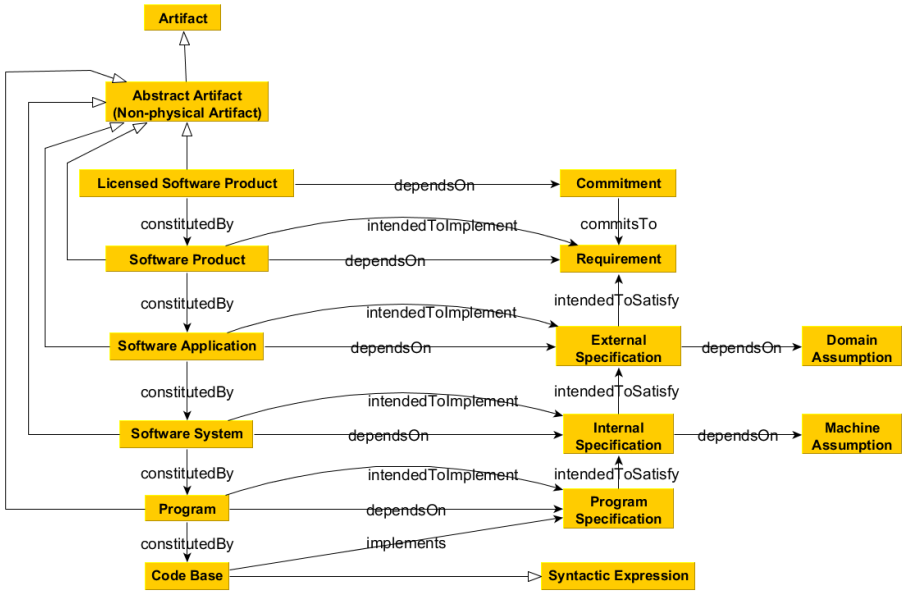


**Fig. 3.** A layered ontology of software artifacts.

**constitutedBy:** We mean here the relation described extensively by Baker [6]. We just assume it being a kind of generic dependence relation that is both asymmetric and non-reflexive, and that does not imply parthood. We can assume here for this relation, the minimal axiomatization present in the DOLCE ontology [16].

**dependsOn:** Among the different kinds of dependence relations (described e.g. in the DOLCE ontology), *dependsOn* denotes in this paper a specific constant dependence relation: if *x* is specifically constantly depending on *y*, then, necessarily, at each time *x* is present also *y* must be present. Again, we can borrow the DOLCE axiomatization for this relation. When this relation holds, being dependent on *y* is for *x* an essential property.

**intendedToImplement:** This relation links an artifact to its specification, as a result of an intentional act. Note that the intention to implement does not imply that the implementation will be the correct one (e.g., bugs may exist).

**intendedToSatisfy:** This relation is proposed to capture the intended role of a specification in the general formula $S \wedge W \vDash R$. That is, *S* is intended to satify *R*, once the assumptions *W* holds.

## 5 Ontology-Driven Software Configuration Management

According to [17], Software Configuration Management (SCM) is "a discipline for controlling the evolution of software systems", and is considered as a core supporting process for software development [18]. A basic notion of any SCM system is the concept of version [19]. The IEEE *Software Engineering Body of Knowledge* states [20] that "a version of a software item is an identified instance of an item. It can be thought of as a state of an evolving item". In the past, the same source distinguished, within versions, between revisions and variants [21]: "A *revision* is a new version of an item that is intended to replace the old version of the item. A *variant* is a new version of an item that will be added to the configuration without replacing the old version". In our approach, these two kinds of version can be described as follows:

**Revision Process. S**uppose that at time *t* we have *p1* constituted by code *c1;* when at time *t'* we replace the code *c1* as the constituent of *p1* by code *c2*, we are not creating a distinct program *p2*, but we are simply breaking the constitution relation between *p1* and *c1*. Thus, at *t'*, *c1* is not a constituent of the program anymore; rather it is merely a code, so that at t' we are still left with the same program *p1*, but now constituted by a different code *c2*.

**Variant Process.** Suppose that we have a software system *s1* ("MST-Finder A"), and we develop a software system *s2* ("MST-Finder B") from *s1* by adopting a new algorithm. Now *s1* and *s2* are constituted by different programs. Of course, *s1* will not be identical to *s2*, since they are constituted by different programs at the same time, and by Leibniz's Law if two individuals have incompatible properties at the same time they are not identical. Indeed, the two software systems may have independent reasons to exist at the same time.

Traditionally, revisions and variants are managed by means of naming conventions and version codes which are usually decided on the basis of the *perceived* significance of changes between versions without any clear criterion (e.g. CVS, SVN). We believe that the layered ontology introduced in this paper can make an important contribution to make this process more disciplined by providing a general mechanism to explicitly express what is changed when a new version is created. This can be simply done by pointing to the software artifact that is affected by the change, and can be reflected by a simple versioning scheme (e.g. v 1.5.3.2: 1 - software application release number; 5

– software system release number, 3 – program release number; 2 – code release number). In addition to this scheme, we can document the *rationale* why a certain software artifact has been changed, by applying the revised reference model of Figure 2 and pointing to the specific source of change.

We believe that this ability to account both for what and why software is changed is essential for software engineering, because managing software and software evolution requires much more than managing code. For example, as Licensed Software Products are based on a chain of dependent artifacts culminating with a computer code, a software management system must be able to manage the impact that changes in the code ultimately have in terms of legal and financial consequences at the level of licensed products.

## 6 Conclusions and Future Work

Based on the work of J&Z and Irmak, we analyzed in this paper the identity criterion of software from the artifectual perspective, extending the analysis to the social nature of software as well. Several kinds of software artifacts have been identified, resulting in a layered ontological structure based on the constitution relation.

Besides clarifying core concepts in the domain of software engineering, our work can also serve as a foundation for software management and evolution. By checking the identity criteria of the software artifacts in different abstraction layers, we can judge the conditions when they keep their identities under changes, or new entities are created. Based on that, a refined versioning methodology and better software versioning control tools dealing with revisions and variants could be developed. As noted several times, traditional tools only focus on code changes. According to our work, software should be consistently expressed and tracked in multiple abstraction layers.

This work is part of a general project on the ontology of software evolution and software change. We hope our work could be used as a foundation for researchers and practitioners working on software maintenance, software project management, software measurements and metrics.

**Acknowledgements.** Support for this work was provided by the ERC advanced grant 267856 for the project entitled "Lucretius: Foundations for Software Evolution" (http://www.lucretius.eu), as well as the "Science Without Borders" project on "Ontological Foundations of Service Systems" funded by the Brazilian government.

## References

1. Osterweil, L.J.: What is software? Autom. Softw. Eng. 15, 261–273 (2008).
2. Eden, A.H., Turner, R.: Problems in the ontology of computer programs. Appl. Ontol. 2, 13–36 (2007).
3. Oberle, D.: Semantic Management of Middleware. Springer, New York (2006).
4. Irmak, N.: Software is an Abstract Artifact. Grazer Philos. Stud. 86, 55–72 (2013).

5. Lando, P., Lapujade, A., Kassel, G., Fürst, F.: An Ontological Investigation in the Field of Computer Programs. In: Filipe, J., Shishkov, B., Helfert, M., and Maciaszek, L. (eds.) Software and Data Technologies SE - 28. pp. 371–383. Springer Berlin Heidelberg (2009).
6. Baker, L.R.: The ontology of artifacts. Philos. Explor. 7, 99–111 (2004).
7. Wirth, N.: Algorithms+ data structures= programs. Ser. Autom. Comput. (1976).
8. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. Proceedings of the 17th international conference on Software engineering. pp. 15–24. ACM, New York, NY, USA (1995).
9. Zave, P., Jackson, M.: Four dark corners of requirements engineering. ACM Trans. Softw. Eng. Methodol. 6, 1–30 (1997).
10. Gunter, C.A., Jackson, M., Zave, P.: A reference model for requirements and specifications. Software, IEEE. 17, 37–43 (2000).
11. Wikipedia: Application software, http://en.wikipedia.org/wiki/Application_software.
12. Mezrich, B.: The Accidental Billionaires: The Founding of Facebook : a Tale of Sex, Money, Genius and Betrayal. Anchor Books (2010).
13. Wikipedia: ELIZA, http://en.wikipedia.org/wiki/ELIZA.
14. Nardi, J.C., De Almeida Falbo, R., Almeida, J.P.A., Guizzardi, G., Ferreira Pires, L., van Sinderen, M.J., Guarino, N.: Towards a Commitment-Based Reference Ontology for Services. Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International. pp. 175–184 (2013).
15. Mozilla: Mozilla Public License, http://www.mozilla.org/MPL/.
16. Masolo, C., Borgo, S., Gangemi, A., Guarino, N., Oltramari, A., Horrocks, I.: WonderWeb-D18: Ontology Library. , Trento (2003).
17. Dart, S.: Concepts in Configuration Management Systems. Proceedings of the 3rd International Workshop on Software Configuration Management. pp. 1–18. ACM, New York, NY, USA (1991).
18. Chrissis, M.B., Konrad, M., Shrum, S.: CMMI for Development: Guidelines for Process Integration and Product Improvement. Pearson Education (2011).
19. Estublier, J., Leblang, D.B., van der Hoek, A., Conradi, R., Clemm, G., Tichy, W.F., Weber, D.W.: Impact of software engineering research on the practice of software configuration management. ACM Trans. Softw. Eng. Methodol. 14, 383–430 (2005).
20. Bourque, P., Fairley, R.E. eds: Guide to the Software Engineering Body of Knowledge Version 3.0. IEEE Computer Society Press (2014).
21. Abran, A., Moore, J.W.: Guide to the software engineering body of knowledge. IEEE Computer Society (2004).