DIGITAL ARTIFACTS AS ABSTRACT OBJECTS: AN ONTOLOGICAL COMPARISON TO MUSICAL WORKS

DUYGU AKTAŞ

BOĞAZİÇİ UNIVERSITY

2019

DIGITAL ARTIFACTS AS ABSTRACT OBJECTS: AN ONTOLOGICAL COMPARISON TO MUSICAL WORKS

Thesis submitted to the

Institute for Graduate Studies in Social Sciences

in partial fulfillment of the requirements for the degree of

Master of Arts

in

Philosophy

by

Duygu Aktaş

Boğaziçi University

2019

DECLARATION OF ORIGINALITY

I, Duygu Aktaş, certify that

- I am the sole author of this thesis and that I have fully acknowledged and documented in my thesis all sources of ideas and words, including digital resources, which have been produced or published by another person or institution;
- this thesis contains no material that has been submitted or accepted for a degree or diploma in any other educational institution;
- this is a true copy of the thesis approved by my advisor and thesis committee at Boğaziçi University, including final revisions required by them.

Signature D.A.....

Date 16.08.2019.....

ABSTRACT

Digital Artifacts as Abstract Objects: An Ontological Comparison to Musical Works

Digital objects such as computer programs, web-based programs, social media accounts etc. appear to change enormously. One position is that such changes are only results of some linguistic conventions rather than true changes. This position suggests that digital objects are mathematical entities. Our daily behaviors about digital objects suggest otherwise. We use several software artifacts which undergo changes through time, and we still believe that we use the same product. I argue that digital objects survive through some changes.

My aim is to give an account of the nature of digital artifacts in a way that is consistent with our linguistic and non-linguistic behaviors concerning these objects. I focus on the identity conditions of digital artifacts, conditions under which the identity of a digital artifacts survives through change.

This work is an attempt to explain the relations between digital artifacts in different situations regarding their identity conditions:

1) When the user preface, the function, and the algorithm of two programs are the same but they have different and independent creators, and they are historically and causally independent.

2) When two programs have different algorithms and different texts but they carry the exact same function, and they are created by the same programmer as the same digital object.

3) When a digital object has changes in its user prefaces, or in its algorithm due to updates, and thereby gains or loses some function or features.

ÖZET

Soyut Nesneler Olarak Dijital Eserler:

Müzik Eserlerinin Ontolojik Statüsüyle Bir Karşılaştırma

Bilgisayar programları, internet tabanlı programlar, sosyal medya hesapları v.b. dijital nesneler sürekli değişen varlıklar gibi görünüyor. Dijital eserlerin matematiksel nesnelerden farksız olduğu yönündeki görüşler değişim olarak tarif ettiğimiz durumların dil kullanım alışkanlıklarımızdan kaynaklandığını ve bu nesnelerin gerçek bir değişim geçirmediğini savunuyor. Dijital nesnelere ilişkin gündelik pratiklerimiz ise bu nesnelerin belirli değişiklikler sonrasında varlıklarını korudukları yönünde. Kullandığımız yazılımlar ve programlar zamanla farklılaşıyor, gelişiyor biz yine de aynı programları kullandığımızı düşünüyoruz.

Bu tezde amacım dijital eserlerin ontolojik konumuna yönelik, dilsel alışkanlıklarımız ve davranışlarımızla örtüşen bir savunuda bulunmaktır. Odağım dijital eserlerin hangi koşullar altında geçirdikleri değişikliklere rağmen özlerini koruduklarını belirlemeye ve böylece bu nesnelerin özdeşlik koşullarını açıklamaya yöneliktir. Bu tez digital eserlerin özdeşlik koşullarını ilgilendiren aşağıdaki gibi durumlarda ne tür ontolojik ilişkileri olduğunu belirlemeye yöneliktir:

1) Tarihsel ve nedensel olarak birbirlerinden bağımsız, farklı kişiler tarafından üretilen iki programın kullanıcı arayüzleri, algoritmaları, ve işlevleri aynı olduğunda,

2) Aynı programcı farklı algoritmalar ve metinler kullanarak tamamen aynı işlevi gören programları aynı program olmalarını amaçlayarak ürettiğinde,

3) Bir dijital eser kullanıcı arayüzünde, algoritmalarında güncelemeler sonucu yeni özellikler ya da işlevler kazanıp kaybettiğinde.

ACKNOWLEDGMENTS

I thank my advisor Nurbay Irmak for his support, for his invaluable feedbacks to this thesis, for his immeasurable contribution to my education; my committee members Sun Demirli and Jack Woods for the critique and feedback they offered; my close circle and favorite persons on earth —Balam Nedim Kenter, Öznur Şahin, Yeşim Aktaş—for being there unconditionally when I most needed. Especially to Balam who read and edited my work at every point of my master's program; my parents Nevim Aktaş, and Adem Aktaş for their support and encouragement.

DEDICATION

This thesis is dedicated to my favorite person, Balam, who believed in my work when I found it impossible to believe in it myself.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

In this thesis, I attempt to investigate the nature of digital artifacts such as computer programs, web-based programs, videos on the internet, social media accounts etc. In our daily lives we interact with these digital artifacts very often. We use them for various reasons and with different intentions. However, there is little research about the nature of digital objects or their existence conditions (Moor, 1978; Suber, 1998; Colburn, 2000; Irmak, 2012; Wang et al., 2014).

My aim is to give an account of the nature of digital artifacts in a way that is consistent with our linguistic and non-linguistic behaviors concerning these objects. I focus on the identity conditions of digital artifacts, the conditions which enable us to identify these artifacts as the same or different objects, and conditions under which the identity of a digital artifact survives through change. I will argue that digital artifacts are abstract objects which are created by their producers with certain intentions. They do not have spatial properties, and they can have various instances which are located in different places at the same time. Moreover, I will argue that it is possible for digital objects to be modally and temporally flexible which means these artifacts can change their matter, and they can survive through these changes. Furthermore, digital objects could have been produced with different matter, or they could have been produced at different times.

Throughout this thesis, I argue for the existential dependence claim for abstract artifacts. This claim has two meanings. The first one is that the producers of

a certain object are causally responsible for the production of that certain object (Thomasson, 2009, p. 194). In the case of digital artifacts, the objects which are produced are abstract objects. Since creation seems to require some kind of causal interaction between the creator and the created object, the claim that we can causally interact with abstract objects prima facie does not seem plausible. This view is called creationism about abstract objects. I will merely claim that it is possible for abstract objects in general, and digital artifacts in particular, to be created by their producers. The second sense of the existential dependence claim concerns the nature of artifacts. Following Thomasson's argument on artifacts, I will argue that not only the existence of digital artifacts depends on their producers, but also that the nature of these artifacts (like all the other artifacts) depends on their makers' intentions to make that artifact as it is (Thomasson, 2009, p. 198).

The existential dependence argument, at least for abstract artifacts, implies "magical modes of creation": the idea that we create abstract artifacts like fictional characters merely by thinking about them (Thomasson, 2009, p. 196). Since abstract objects do not come into existence by modifying a certain kind of matter, they seem to emerge from nothing. Thomasson thinks that the existential dependence argument does not commit someone to the magical modes of creation claim. She thinks that instead of committing ourselves to magical modes of creation which means claiming that mere human intentions are sufficient to create an artifact, we need a new methodology. We should identify the conditions that determine the successful application of a given term or a given object (Thomasson, 2009, p. 197).

I will examine two accounts on the nature of digital artifacts which attempt to provide a new methodology to determine the identity conditions of digital objects.

One account suggested by Irmak, argues that digital objects are certain kinds of abstract artifacts which are created with certain intentions and which can be destroyed. Digital artifacts, from an ontological point of view, are more like musical works rather than mathematical entities. The dependence relations between digital artifacts and their algorithms, executions, text copies, and producers are similar to the dependence relations between musical works and their composer, sound structures, scores, and performances. Moreover, there are many studies that focus on the artifactual nature of musical works as abstract entities, and their identity and persistence conditions (Levinson, 1980; 2001; Predelli, 2001; Howell, 2002; Caplan, and Matheson, 2004). Hence, Irmak provides an account of digital artifacts with a comparison to musical works.

According to another account which was suggested by Wang, Guarino, Guizzardi, and Mylopoulos (2014), digital objects are complex artifacts. During the creation process of a software product, many kinds of artifacts are brought into existence such as programs, software systems, software products, etc. Each of these artifacts are inter-related by layered constitution relations. The term layered constitution relations means that during the programming process a programmer produces various different digital artifacts such that some of them belong to different kinds, and some kinds constitute others. This is the reason they call their view a layered ontology.

I argue that both of these accounts provide some valuable insights on the nature of digital artifacts. However, the first account does not attempt to provide an explanation about the conditions under which digital objects survive through changes, or under which changes new entities are created. The second account faces

some serious problems arguing for such conditions. This thesis is an attempt to explain the relations between the digital artifacts in different situations regarding their identity conditions:

1) When the user preface, the function, and the algorithm of two programs are the same but they have different and independent creators, and they are historically and causally independent.

2) When two programs have different algorithms and different texts but they carry the exact same function, and they are created by the same programmer as the same digital object.

3) When a digital object has changes in its user prefaces, or in its algorithm due to updates, and thereby gains or loses some function or features.

In what follows, I will first argue that Irmak's account does not provide us with a tool to allow for changes in digital artifacts. Second, I will argue that Wang et al. attempt to provide a tool to explain that digital artifacts can survive through certain changes, however they fail at their attempt. The main reason for this is that the constitution relation which Wang et al. have adopted is not applicable to digital artifacts, and its application leads to from a metaphysical point of view odd conclusions. Finally, I will suggest another constitution view, which is argued by Simon Johan Evnine, to apply to digital artifacts and to examine how the application of this view explains the nature of digital artifacts. I will argue that the application of this account would remedy the problems that Wang et al.'s view struggles with.

CHAPTER 2

IRMAK'S ACCOUNT OF SOFTWARE, AND THE PROBLEM OF CHANGE

Most of the research on software focuses on the dual nature -concrete and abstract- of digital artifacts, especially that software (Moor, 1978; Suber, 1998; Irmak, 2012; Wang et al., 2014). Moor argues that the distinction between software and hardware is not ontologically important (Moor, 1978). Moreover, he argues that software can be identified with a machine system mainly with that whereby the initial programming of a given software occurs. However, the software and the machine system have different persistence conditions. One can destroy the machine system which Moor is eager to identify with software, but software can survive this destruction through some copies of itself. Suber argues that everything which has a pattern is software (Suber, 1998). This argument is too strong, and it begs the question about the nature of software because it merely suggests that everything is software (Irmak 2012, p. 58). Hence, neither Suber's not Moor's account provides an explanation of the nature of software. The nature of software is argued by Irmak (2012), and later following Irmak by Wang et al. (2014). In this chapter, I will focus on Irmak's argument.

Irmak argues that software is a certain kind of abstract artifact which is created with certain intentions and which can be destroyed. Software does not have spatial properties, rather it has temporal properties; it comes into existence at a time and might cease to exist at another time. Given that software has temporal properties and can be created, it is distinct from platonic entities which are eternal abstract objects like mathematical objects (Irmak, 2012, p. 56). Irmak's account avoids

certain errors such as identifying software with its instances of the machinery that runs it, or saying that everything is software. Moreover, the account focuses on counting software as the same or different objects according to our linguistic and non-linguistic practices.

I begin with the distinction between software and their code and algorithm since these are the generally accepted candidates that are considered to be identical with software. Recent research focusing on the identity of software agree that software, and its code, and its algorithm are distinct entities (Irmak, 2012; Wang et al., 2014). Irmak argues that identifying software with its code or its algorithm is fallacious because they are different from software in some important ways. An algorithm is an eternal object similar to mathematical entities such as numbers and sets whereas software, as he argues, is an abstract artifact (Irmak, 2010, p. 59). Furthermore, the same software might have been produced by two different algorithms; and having the same algorithm does not necessarily guarantee having exactly the same software (Irmak, 2012, p. 61). The code is a language-dependent abstract object which is also distinguishable from software because the very same software might be written by different codes in different programming languages (Irmak, 2012, p. 61). According to Wang et al., another reason code to might not be identical with its software is that software is argued to be an intentional product of its producers while code is not necessarily an intentional product. Someone can produce a code accidentally by adding a line or deleting some lines from a code-base. On the contrary, software should be created with certain intentions by its producer(s) (Wang et al., 2014, p. 324). Other possible candidates that could be identical with software, like the execution of software or copies of software, cannot be identical with

software either due to all these candidates having different persistence conditions from software (Irmak, 2012, p. 60).

Irmak thinks that although software is not identical with its algorithm, it is closely related to it. In order to find the relation between software and its algorithm, he applies Levinson's tripartite account of identification of a musical work to software. Levinson thinks that a musical work is identical to the sound and performance structure which is indicated by its composer at a given time (Levinson, 1980, p. 20). Irmak substitutes sound structure with algorithm, which are both eternal abstract objects and modifies the view as software being an abstract artifact which is produced by its producers at a given time. He thinks that this identification satisfies the creatability requirement by saying that software did not exist before its producers produced it (Irmak, 2012, p. 69).

However, Irmak finds this identification problematic (Irmak, 2012, p. 65). He thinks that any successful view on the nature of software must take into account that software survives through changes in its algorithms or its code (Irmak, 2012, p. 65). This view cannot account for any change to the algorithm because it offers a strict identification criterion concerning the algorithm, the producer(s) of the algorithm, and the time that the software was produced. Software cannot survive through changes in these three respects. As Irmak also highlights, our social behaviors towards digital objects conflict with this part of the conclusion (Irmak, 2012, p. 71). Irmak concludes that software is an historical kind of abstract artifact which is produced by its programmers with certain intentions (Irmak, 2012, p. 68). Although this conclusion does not provide any tool to account for software identity over changes in their algorithm or code, it does provide some important insights. Instead

of identifying algorithm with software, it suggests that producers of software and certain intentions of these producers can be in locus of software identity. Moreover, it makes an important claim about the creatability of software by pointing out the creation time of the software. In this account, software is rigidly historically dependent on its producers and certain intentions of these producers. This means that software is dependent on a particular individual by whom it has been produced, and the time at which it has been produced. Whenever software exists, its producers also exist either prior or at the time of software production. This point has an important role distinguishing between very similar software such as two software programs that are produced by different and independent creators and which are historically and causally independent from each other, yet they have the same algorithm (Irmak, 2012, p. 70). Two objects historically and causally independent from each other means that they can come into existence separately and independently from each other. Neither of them pre-requests the existence of the other nor is it caused by the other. Although these two programs might seem to us as identical they are different from each other. Hence, this account provides identity conditions which can explain the condition that I described above as item 1.

Conditions described as items 2 and 3 focus on the relations between software and its algorithm. The idea is that software is closely and strictly related to its algorithm. A firmer version of this idea is that software is rigidly constantly dependent on its algorithm; if we were to refuse that software is identical to its algorithm, this would not correspond to our usage of the term software. In each of these cases, whenever there is a change in the algorithm of software, necessarily there is another software. Moreover, in the case that software is rigidly and

constantly dependent on its algorithm which means that whenever software exists it, necessarily, has the particular algorithm that it actually has.

Following Irmak, Wang et al. argue that software is a complex artifact which cannot be identified with its algorithm, code, or execution. During the creation process of a software product, many kinds of artifacts are brought into existence such as programs, software systems, software products, etc. Each of these artifacts is distinguishable from one another by its essential properties and its encoded intentions (Wang et al., 2014, p. 322). Furthermore, these artifacts are inter-related by layered constitution relations (Wang et al., 2014, p. 322). Wang et al.'s account uses various dependence relations and a constitution relation to explain the nature of digital artifacts. In the following chapter, I will examine their account. First, I will focus on dependence relations in their account, and argue that they have some, ontologically speaking, odd claims. Second, I will examine the constitution view they use. They argue that, during the production process of a digital artifact, a programmer produces various digital artifacts some of which belong to different kinds, while some different kinds of artifacts constitute others. I will argue that the constitution notion they have adopted is not applicable to their view for various reasons. Moreover, the application of their adopted constitution notion ends up with a problematic metaphysical conclusion.

CHAPTER 3

WANG ET AL.'S VIEW: A LAYERED ONTOLOGY OF DIGITAL ARTIFACTS

Wang et al. argue for various dependence relations which hold between software artifacts and their essential properties, or their producers, or a community who understands the essential properties of these software artifacts. In Wang et al.'s view during the programming process a programmer produces various different digital artifacts. Some of these artifacts belong to different kinds, and some kinds constitute others. They call this notion a layered ontology of software. Although Wang et al. accept that software is an abstract object in their layered ontology, they argue that a kind of constitution relation is possible between the objects that are produced during the creation process of software. They adopt Baker's notion of artifacts and her view on constitution. The main argument is that a new kind of entity comes into being when a certain group of entities of a given kind are in certain circumstances (Baker, 2014 as cited in Wang et al. 2014, p. 325). They accept that artifacts are intentionally created by their producers to carry an intended proper function. In Baker's view, artifacts are intentionally designed and produced by their producers to perform a proper function. For example, a pencil is produced to inscribe and draw on papers. The proper function of a pencil is being a tool to write and draw on papers. Moreover, artifacts carry their proper function essentially whether they are used for their proper functions or not (Baker, 2004, p. 102). In the light of Baker's view, Wang et al. set three identity criteria for software: software has a certain purpose, it is created intentionally, and it is recognizable by some individuals (Wang et al. 2014, p. 322). In the following, I will first examine these three criteria, and the dependence

relations they argued for to set up these three criteria (3.1). I will use these analyses in the rest of this thesis. Second, I will examine their account on different kinds of software artifacts, and their adopted constitution view (3.2).

3.1 Identity criteria and dependence relations

The first identity criterion Wang et al. argue for is that software has a certain purpose. During the production process of a software, a programmer produces various kinds of software artifacts. Each kind of software artifact has a different proper function (Wang et al., 2014, p. 324). Proper functions of software artifacts are determined by their producers. For example, one of the artifacts they argue for is a program. The proper function of a program is to produce pre-determined results in a particular way on a computer (Wang et al., 2014, p. 324). The intended proper function of an artifact is anchored in its essential property which means that intentions of a producer carried by the essential property of an artifact (Wang et al., 2014, p. 324). For instance, a programmer decides to produce a program. She intends to arrive at certain results on a computer. In order to produce such results, she decides the effects she wants to reach at, and describes how to reach these effects in a way in which that can be processed by a computer. Her decisions and descriptions regarding her program are carried by a program specification which is the essential property of a program. The first identity criterion of this view relies on two different dependence relationships. Since the intentions of a software artifact are determined and anchored by their producers, artifacts and their producers have a dependence relation. Given that the proper function of an artifact is carried by its essential property, an object and its essential property have a dependence relation too. I will

explain their view on essential properties and dependence relations regarding these properties later in this sub-chapter.

A software artifact being historically dependent on its producer means that a software artifact requires a producer in order to come into existence (Wang et al., 2014, p. 322). This historical dependence relation means that a software artifact did not exist before its producer(s) produced them. In a sense it makes an existential claim on software artifacts which is whenever there is a software artifact, it is necessarily produced by a producer. In Wang et al.,'s argument there is no specification as to whether this is a rigid historical dependence or a generic historical dependence. It might be the case that a software artifact is rigidly dependent on their producer which means that a software artifact is necessarily dependent on a particular individual by whom it has been actually produced. Also, it might be the case that a software artifact is generically dependent on its producer which means that an artifact requires a producer (and the productive activities of a producer as I will explain later in this chapter) to come into existence. However, the artifact is not necessarily produced by a particular individual by whom it has been actually produced. Since the same artifact could have been produced by another producer, then it seems that what Wang et al. have in mind is a generic historical dependence. They do not have an argument about this issue. They only suggest that a software artifact is historically dependent on its producer.

The recognizability criterion means that there should be some documentation of a given software artifact which is understandable by some rational beings, and that there should be such rational beings to understand these documentations. According to Wang et al. this is an identity criterion for software artifacts (Wang et al., 2014, p.

322). Software artifacts are generically constantly dependent on a community that recognizes and understands their essential properties (Wang et al., 2014, p. 330). A generic constant dependence on a community means that the members of this community can change. However, in order for a software artifact to exist, there must necessarily be a community to understand the essential properties of the given software artifact. This is the reason Wang et al. think that a software artifact necessarily be documented in a way in which its essential property can be understood through this documentation. An example for a documentation of a software artifact is a written copy of its code which can be properly recognized and understood by a community (Wang et al., 2014, p. 330). A code is the first entity a computer producer creates during the programming process. As a reminder, a code is a language-dependent set of instructions.

The problem is that Wang et al. think that the criterion concerning recognizability of a software artifact is an identity condition for an artifact. But it is actually a persistence condition which means it is an explanation about under which conditions a software artifact continues its existence rather than under which conditions it comes into existence. A software artifact can endure via some documentations of itself like a copy of its code in a computer. Furthermore, a software artifact can endure via some rational beings who can understand and remember the nature of the given software artifact. However, a software artifact can come into existence without a community of people or even without any documentation of itself. For example, a programmer with a perfect eidetic memory can produce a program in her mind. Assuming that no one else knows about the program, the program which she has created exists in her mind as long as she

remembers the program. In order for a software artifact to come into existence there is no need for a (physical) documentation[1] of the program, and there is no need for a community to understand the program. In this case, the program our programmer creates only exists in its producer's mind. If she forgets her program, assuming that there is no documentation of her program, and no one else knows about the program; then the program ceases to exist. Even though the program ceases to exist under certain conditions, it has been created without satisfying the recognizability criterion in its programmer's mind. The reason for this is that the recognizability criterion is about the persistence conditions of a software artifact. A software artifact can survive via one of its copies, or a person who can remember the artifact even if its producer dies.

The way Wang et al. argue for their recognizability criterion is problematic for cases regarding private production and usage of a software artifact. They think that this criterion necessarily excludes the software artifacts which are produced in privacy and used only by their producers, however they do not provide a sufficient argument for this exclusion. For instance, let's assume that our programmer has produced a program, and then she ran the program in her computer. She only produced this program for her own use, and again no one else knows about the program. In this case there is a proper documentation of the program in the programmer's computer -namely, the code of the program which is written in her computer. However, Wang et al. still think that in this case, the constitution relation that holds between this program and its constituent exists only in the programmer's

[1] In this case, our programmer has a mental documentation of her program or a mental token of her program.

mind (Wang et al., 2014, p. 329). They accept in a footnote that their view excludes some software artifacts which are produced for personal usage in private. But, they think such objects belong to a limited case, and decide to bite the bullet (Wang et al., 2014, p. 322). The exclusion of such cases might not be problematic for software engineering processes. However, it is problematic for the ontology of software artifacts because they accept that a written copy of the code of a software artifact is a documentation of the given software artifact under the condition that it can be properly recognized and understood by a community (Wang et al., 2014, p. 330). On the other hand, they insist that the program exists only in its producer's mind, even though she runs the program in her computer. It seems to me that a copy of a code in a computer is a sufficient documentation. In the case of a private usage involved in the private production process, their claim about the necessity of a documentation and recognizability seems to be satisfied. Even if the producer dies right after she runs the program in her computer, the program would have a copy of its code in her computer. There is a possibility that the program can be discovered and can be recognized by another person through its code. I think the recognizability criterion is not an identity criterion as they claim. Moreover, I think they do not provide a sufficient reason for excluding private production cases.

The most important dependence relation they argue for is the constant dependence relation between a software artifact and its essential properties which means whenever a given software artifact exists, it necessarily has an essential property which belongs to its kind (Wang et al., 2014, p. 323). They argue for different kinds of software artifacts, and each kind of software artifact has different essential properties. As I mentioned above, these essential properties carry the proper

function of the given kind. For example, the essential property of a program is its program specification. A program cannot exist without a program specification. Although they argue explicitly only for a constant dependence between an object kind and its essential properties, their argument implies a rigid constant dependence which means that whenever an object exists it necessarily has the particular essential property that it actually has. For example, the essential property of a program is its program specification. A program specification includes three components: a data structure which is a particular way of organizing data for effective data usage in a computer, an algorithm which implements a certain function that is intended to create a desired change within such data structure, and a procedure that produces changes through manipulation of the data structure (Wang et al., 2014, p. 324). According to Wang et al. a program is constantly dependent on its program specification. They explain this constant dependence relation in two ways. That a program constantly depends on its program specification means that whenever a program exists, its program specification necessarily exists. Moreover, a program is being constantly dependent on a program specification, in their account, means that a program specification is the essential property of a program (Wang et al., 2014, p. 331). However, in their account, whenever there is a change in the essential property of an object, there is a new object which is distinct from the previous object. For example, whenever the program specification contains a different algorithm, there is a new program because the algorithm is a component of a program specification. Although they do not explicitly argue for this, it seems to me that, in their view, objects are

rigidly constantly dependent on their essential properties which means whenever there is an object, it necessarily has the particular essential property that it actually has.

Moreover, in the way Wang et al. argue, it seems to me that the essential property of an artifact is created by their producers. I will not argue for the creatability of properties, for two reasons: I will not use this argument in my thesis, and I will not use this argument for or against their view. I think my arguments against their view hold regardless of whether the producer of these artifact creates the essential properties of the products they argued for or not. Second, as I explained above they argue that objects and their essential properties have a rigid constant dependence relation. Assuming that essential properties cannot be created, then they can still claim that their objects rigidly constantly depend on what they described as an essential property of the given kind.

My purpose here is to show that their view implies the creatability of the essential properties, or the creatability of the entities which they describe under the term “essential properties”. Wang et al. describe the producing process of a program in the following way: when a producer decides to produce a program, the first thing she needs to do is to determine the essential property of this program which is a program specification. As a reminder, a program specification includes three components: a data structure, an algorithm, and a procedure. Amongst these components an algorithm is an eternal abstract entity which means it is like mathematical entities, and it is not a product of people. The programmer starts to produce a program by defining the data structure which is a particular way of organizing data for effective data usage in a computer that the program will use. She

writes a code to implement an algorithm. Again, a code is a language-dependent set of instructions which deciphers how the given algorithm works step by step in a computer language. The producer of the program works on these data, to produce the desired changes by manipulation of the data structure through an algorithm and a code. In the way in which Wang et al. describe the process, it seems to me that the work of a programmer is somehow transitive to the program. The programmer seems to interact with an algorithm and a code base. It is through these interactions that she produces an abstract artifact, a program. As I mentioned above, since creation seems to require some kind of causal interaction between the creator and the created object, the claim that we can causally interact with abstract objects prima facie does not seem plausible. I assume that abstract artifacts such as musical works, fictional characters, and digital artifacts are the products of their producers. If there were not certain creative activities of their producers, there would not be such artifacts either. This position has been defended by many philosophers (Thomasson, 1999-2009; Evnine, 2009; 2016, Levinson, 1980; 2001; Caplan and Matheson, 2004). I will not focus on this point right now. I will assume that some abstract objects are the product of certain human intentions and activities which means these artifacts would not come into existence without certain activities of their producers. Assuming that there are abstract artifacts, I think what Wang et al. describe under the label of "essential properties" can also be produced. If they insist that their usage of the term essential properties, ontologically speaking, refers to properties; then in addition to the creatability of an abstract object, they need to rely on the creatability of a property, more specifically the creatability of essential properties. They can also argue that a programmer produces a software artifact by choosing certain properties as the

essential properties of the given object. As a result of her choosing activities she brings into existence a software program. Therefore, I think the creatability claim their argument implies is not problematic.

The problem occurs because they use essential properties of an artifact as a tool to distinguish different kinds of artifacts. For example, the essential property of a code base is its syntactic structure, and the essential property of a program is its program specification. These two objects belong to different kinds because they have different essential properties. However, they also argue that some of these different kinds of objects overlap with regards to their essential properties. Their view runs into serious problems regarding the distinctions they make between different kinds of objects and the relations between these different kinds of objects. According to Wang et al. the relation between these different kinds of objects is constitution. In the next sub-chapter, I will focus on the constitution view which they have adopted. I will examine how they define the constitution relation, and how this relation is applied in their view.

3.2 A constitution relation

In Wang et al.'s view there are two fundamental ontological relations: dependence relations, and constitution relation. I have explained the dependence relation they use in the previous sub-chapter. Now, I will focus on the constitution view that they have applied to software artifacts. I will have two main arguments against their account. First, I will argue that the constitution relation they have adopted does not hold amongst the objects in their account. Second, even if this constitution relation were

applicable to their view, it would have some serious metaphysical problems with regards to identifying software objects such as violation of transitivity of identity.

As I mentioned above, Wang et al. argue for the existence of different kinds of software artifacts, and they argue that some of those artifacts constitute others. Their argument encompasses objects such as a code base, a program, a software system, etc. Each of these objects belongs to different kinds, and a certain kind of object constitutes another kind of object. For example, a code base constitutes a program, and a program constitutes a software system. At the end, all these different software artifacts constitute a software product. A software product is the closest entity to our usage of the word "software".

First of all, I will briefly explain the process of the production of a software product. After this brief explanation, I will examine how Wang et al. individuate different software artifacts in their account, and the relations between these software artifacts. In their view, the production process of a software product occurs in a somehow cumulative way and by following a somehow linear method. Meaning that a programmer produces a software artifact which belongs to one kind, then produces another object which belongs to another kind. This second object is a new kind of software artifact which is constituted by the previous object which itself belongs to a different kind. I will explain the production process in detail throughout this chapter. For a brief explanation of the process: When engineers decide to produce a digital artifact, they intend to satisfy some functions through this artifact. They decide an intended function specification for their software artifact. This intended function specification contains the desired function of the program, which is the purpose of the program, and the algorithm by means of which its determined functions are

supposed to be implemented. The identity of the program is constantly (and as I mentioned above in 3.1 I think also rigidly) dependent on this intended function (the essential property of a program.). Hence, any change to the algorithm brings into being a new program. A program can survive through some changes as long as it has the same algorithm: it can be produced in a different programing language, its code base can change, and the software program can survive through these changes (Wang et al., 2014, p. 326). However, when the algorithm of a given program changes, a new program comes into existence. The group of programs which was brought into existence during this altering process constitutes a new kind of entity: a software system. Furthermore, under certain circumstances different programs can constitute the same software system (Wang et al., 2014, p. 325). Similarly, different software systems can constitute the same software application which is a different kind of software product, and different software applications can constitute the same software product which is the closest object what we call software in our daily uses (Wang et al., 2014, pp. 326-330). They accept that each of these software artifacts presents a different layer in the production process of a software product. Assuming that a programmer completed all these production processes, these different software artifacts constitute a software product at the end of the producing process. This layered ontology promises a more flexible account concerning the algorithm of software.

Assuming that Wang et al.'s argument regarding software objects hold, the same software product can consist of certain artifacts brought upon by layered constitution relations. It seems to me that when software products have different algorithms and different texts but they carry the same exact function, provided that they are created

by the same programmer to be the same digital artifact, they are considered to be the same software product as well (item 2, which I set at the beginning). The criterion for this is that they should be intended to carry the same software requirements. Software requirements are kinds of definitions about functions that can be carried by a certain software product and certain features that the software product can have. Given that software programs which I described in item 2 carry the exact same function, they might be considered to be the same program.

However, the account faces with some serious problems in terms of explaining how the constitution relation holds amongst software artifacts. One of the problems in their account is that the distinction between objects which belong to different kinds is not clear. Moreover, their argument relies on these distinctions because they argue that the constitution relation holds between object which belong to different kinds. They adopt Baker's constitution view that referenced in the previous sub-chapter. According to Baker, a new kind of entity comes into being when a certain group of a given kind are in certain circumstances (Baker, 2004 as cited in Wang et al., 2014, p. 325).

In Wang et al.'s view, when a programmer decides to create a software artifact, the programmer should start with a program because a program will constitute a higher-level object, later in the programing process. The first task of the programmer is to define a program specification. The first criterion, the purpose of a program, is defined by its program specification. Since the identity of a program is tied to its intended specification, the program specification should exist prior to the program (Wang et al., 2014, p. 324). As I mentioned above, a program specification includes three components: a data structure which is a particular way of organizing

data for effective data usage in a computer, an algorithm which implements a certain function that is intended to create a desired change within such data structure, and a procedure that produces changes through manipulation of the data structure. Any change within a program specification results with a different program because the program is rigidly constantly dependent on its program specification. The most important component of a program specification is an algorithm. The nature of algorithms and how to individuate them is important to understand Wang et al.'s claims about individuating software artifacts. For this reason, I explain the issue, before I introduce their view on the constitution.

In the following sections of this thesis, I will discuss two specific algorithms: Selection Sort and Merge Sort algorithms. They are both sorting algorithms which means they are both used for the purpose of organizing variables in an ascending order. For instance, if a programmer decides to produce a program which can sort ticket prices for movie theaters to find the cheapest theater, she starts by defining the program specification. The first thing she needs to decide is the algorithm she will use. In the case that our programmer wants to produce a ticket sorting program, she needs to use a sorting algorithm. Every algorithm contains a different set of instructions. Even though different algorithms end up with the same result, the way they get the result is different from each other. Each algorithm uses a specific set of steps to arrive at the desired result. In the case of our ticket sorting program, our programmer first collects the information of ticket prices. For the purpose of brevity, I assume she considers only six movie theaters. Ticket prices of these movie theaters illustrated by set A. Assume that each number represents a ticket price in a used money currency, our set looks like this: A {9, 12, 27, 23, 8, 13}. Now, our problem

is to organize this set in an ascending way in order to find the cheapest ticket price. Sorting algorithms produce an array from the members of their defined set, in our case this set is A. A data structure is the way in which the data regarding our set is represented in the memory of a program. A series of items which constitutes our set is called an “array”. Our algorithms function in order to provide us an organized array by organizing numbers in our set in an increasing order.

The Selection Sort algorithm works in a way such that it first finds the smallest number in the set by scanning every member of the set (Manber, 1989, p. 130). It starts with the first number in the set which is 9 in our example. Since 9 is the only number the algorithm has seen so far, it is the smallest number at the first step. The next number that the algorithm scans is 12 which is greater than 9, hence 9 remains as our smallest number in our array. Now, the algorithm scans the third number in the set A, sees 27 which is also greater than 9, hence our smallest number remains as 9. When the algorithm scans 8, it swaps 8 at the first row in the array. Our new smallest number becomes 8. Then, the algorithm scans the last number 13 which is greater than 8. At the end of all these steps the algorithm completes one loop which means our set is scanned once from the beginning to the end. The algorithm arrives at the smallest number in our array which is 8. The first row in our current array is occupied by 8. Now, in order to complete the array, which means organizing all numbers we have at hand in an ascending order, the algorithm repeats the same steps five more times. Each time, it determines the next smallest number in order to write the array (Manber, 1989, p. 130). In its second loop, our algorithm finds the second smallest number, which is 9, by scanning the rest of the set again in the same manner that it has scanned in its first loop. Then, it writes the second smallest

number after the first smallest number. Now, our array has two numbers organized in an ascending order, which are 8 and 9. The same loop repeats itself until the point where there is no number remaining in our unorganized set. The final version of our array looks like this: 8-9-12-13-23-27. In our example, each loop contains six steps, because we have six members in our unorganized set. Moreover, it runs the same loop in order to write each row, so it runs the same loop six times. Until the algorithm completes its task and arrives the end result; it goes through thirty-six steps.

The Merge Sort algorithm works in a different manner. The first step of the Merge Sort algorithm is to check if the size of our unorganized set is greater than one. If the size of the set is greater than one, the algorithm divides the set into two halves (Manber, 1989, p. 131). Each of these halves occupy one side. Again, we have set A at hand, which is: A {9, 12, 27, 23, 8, 13}. Since our set contains six members, the algorithm divides it into two arrays of size three. At the end of this step we get two arrays: (9-12-27) on one side of the row, and (23-8-13) on the other side of the row. These two arrays are still bigger than size one. Hence, they get divided again we have the following arrays: (9-12), (27) on one side, and (23-8), (13) on another side. The algorithm divides our two membered arrays once more. Finally, our original set is divided into arrays of size one. Now, each of our original set numbers stands as six different arrays of size one which looks like this: (9), (12), (27) on one side and (23), (8), (13) on another side. After this point, the algorithm is ready to merge these numbers together. Merge Sort algorithm merges two arrays simultaneously and separately (Manber, 1989, p. 131). The one array contains (9), (12), (27), another array contains (23), (8), (13). The algorithm starts with the first two numbers of the

left side which are 9 and 12. The algorithm compares only these two numbers. It decides that 9 is smaller than 12, and writes 9 at first. Next, it has only one number at hand which is 12, because it has selected only 9 and 12 in the previous step. For this reason, it writes 12 at second. The algorithm has successfully merged two numbers in an array and arrives at a pair which is (9-12). Now it compares the remaining number of the left side with the first number in the ordered pair. At this step our numbers at hand are 9 and 27, and 9 is smaller than 27. Hence, the algorithm writes 27 at the end of the previous pair and arrives at an ordered array in size three which is: (9-12-27). The exact same steps are followed for the right side of the array which contains (23), (8), (13) simultaneously with the left side of our array. The right side arrives at its own ordered array which is (8-13-23). Finally, the algorithm merges these two arrays into one array (Manber, 1989, p. 131). First, it chooses the first number of each array which are 9 and 8, compares these two numbers and picks the smallest one, which is 8. It writes 8 first, and then adds 9 next to it. Then it compares 12 and 13, and adds first the smallest one, then another number at hand after the already organized pair: 8 and 9. The end result is the same with the previous algorithm which is 8-9-13-23-27.

These two algorithms Selection Sort and Merge Sort are two distinct set of instructions as it can be seen from the illustrations. They work differently, at different speeds. They are preferable for different kind of problems. For example, sorting for a long set of numbers the Merge Sort is more beneficial than the Selection Sort algorithm because it works on separate tasks simultaneously which makes it quicker than the Selection Sort algorithm. Moreover, if our programmer decides to write another program to find the quickest path between her work and her house, then she needs to use a graph algorithm instead of a sorting algorithm. A graph

algorithm consists of some points and functions for connecting these points together. There are various graph algorithms and various sorting algorithms. The programmer chooses the function of her program and decides what kind of algorithm can satisfy her intentions.

When our programmer makes a choice about an algorithm, she starts writing the code of the program. The code is a written version of these instructions step by step in a computer language. Wang et al. say that the programmer writes the code by implementing an algorithm in a way in which it allows the algorithm to run in a computer (Wang et al., 2014, p. 325). The same algorithm can be implemented in different computer languages. However, each computer language has specific symbols and syntactic rules to write a code. Since Wang et al. argue that a code is rigidly constantly dependent on its syntactic expression, each computer language brings a different code base for the exact same algorithm. According to Wang et al., the code starts to constitute the program from the point at which the programmer decides the code is ready to be tested, and it suffices to implement the program (Wang et al., 2014, p. 325). Therefore, certain conditions to create a program include: intention to create a certain program, to create its essential property—program specification—, and to make a decision on when the aggregate of the program (the code base) suffices to implement the program. I want to highlight that these conditions are parallel to the conditions they set as identity criteria for a software artifact except for the recognizability claim. As a reminder, their identity criteria are the following: a software has a certain purpose, it is created intentionally, and it is recognizable by a community through its essential property. To sum up, a program specification is determined in the following way: The programmer encodes

the specific function of her program by defining a program specification. She defines the program specification by determining the function of the program, the algorithm to use for the specific function she has determined, and the data structure of the program. She writes the code of the program to decipher the instructions of the algorithm to a computer language in order to produce certain effects on a computer, and these certain effects are described in the program specification.

Wang et al. argue that "by an act of creation" the programmer decides that the code constitutes the program as soon as she decides the code is ready to run the program specification (Wang et al., 2014, p. 325). After the programmer decides to run the program through a computer, if the program does not work in the way in which its programmer expects it to, then the program is in fault not the code (Wang et al., 2014, p. 325). It is not clear in their argument whether the programmer can be mistaken about the existence of a new artifact. After the decision to run the program, whether the program functions or not, there is a new program. However, I remind that there is also a recognizability claim for any software artifact and, based on their recognizability claim, Wang et al. think the program exists only in the mind of our programmer. After defining a program specification, the programmer can change the code of the program for developmental or maintenance purposes. Throughout these changes, although she is working with different code bases, as long as she keeps implementing the same algorithm, she is working on the same program. For instance, our programmer uses a code base (c1) to implement the Selection Sort algorithm for her ticket sorting program (let us call this program TickEd), then she replaces the c1 with another code base (c2) to implement the Selection Sort algorithm again. She still works on the same program. The reason for this is that these different code bases

(c1 and c2) constitute the same program, TickEd (Wang et al., 2014, p. 323). In this part, the constitution relation seems applicable to these two software artifacts: code bases and programs.

However, if she decides to change the algorithm, then she creates a new program (Wang et al., 2014, pp. 325-326). To illustrate, if our programmer changes the algorithm of her program and, replaces the Selection Sort algorithm with the Merge Sort algorithm, then we have a different program specification, and therefore, a different program (let's call this program TickEd-B) Our programs TickEd and TickEd-B have the same functions: organizing the ticket prices of movie theaters in an ascending order. According to Wang et al., what these programs have in common is the fact that they are both constituents of a new kind of entity: a software system (X Wang et al., 2014, p. 325). A program and a software system are different kinds of objects because each has different essential properties, and a software system is constituted by a program. These two programs, TickEd and TickEd-B, constitute the same software system under certain conditions I will explain later in this chapter. For now, I assume that the programmer produced the TickEd program, and without producing a software system she changed the algorithm; then she produced another program: TickEd-B. These two programs constitute the same artifact: a software system (Wang et al., 2014, p. 325).

The question is how the constitution relation holds between code bases and a program, or a program and a software system, and/or other digital artifacts. Given that in Wang et al.'s view digital objects are accepted as abstract objects, these objects do not have any spatial properties. However, they adopt Baker's account of constitution, and Baker's account of constitution requires spatial coincidence.

According to Baker, constitution is a relation between the spatially coincident objects of different kinds (Baker, 2007, p. 32). An object and its constituents share some non-derivative properties through this spatial coincidence. Moreover, a constituted object comes into existence with new causal powers (Baker, 2007, p. 37). I will avoid the discussion on whether abstract artifacts have causal powers or not. Instead, I will focus on how the constitution relation is established according to Wang et al., and whether it is possible to apply the constitution relation to abstract entities.

In the section on dependence relations of this chapter, I explained that unless the constitution relation which bounds a program and its code base is documented, the constitution relation only exists in the mind of the programmer. The fundamental point is that they do not argue that the documentation of the program and the program itself has a constitution relation. They argue that a program is constituted by a code which is written by a programmer with an intention to produce a certain program. The problem is that how the constitution relation holds is not clear from their arguments. The first problem regarding this issue is that, as I mentioned above, the constitution relation requires spatial coincidence, and software artifacts do not have spatial properties. The second problem is that the constitution relation which they have adopted holds between different kinds of entities. Wang et al. argue that each object they argued for belong to different kinds, however, how these objects belong to different kinds is ambiguous in their argument. As a result of this ambiguity, how the constitution relation is supposed to work is also ambiguous.

Baker describes constitution relation as “genuine unity” (Baker, 2007, p. 37). She thinks that although the constituter and the constituted object are not identical, they are not separate and independently existing individuals. In Baker’s account,

constitution relation holds when some aggregates come together under certain circumstances in which a new kind of entity comes into existence (Baker, 2007, p. 32). This new kind of entity and its aggregates are necessarily different "primary-kinds" (Baker, 2007, p. 32). The term "primary kind" is a denotation on the most fundamental feature of an object. Objects that have different primary kind properties have different persistence conditions. According to Baker, primary kind is essential to every object (Baker, 2007, p. 33). In the case of artifacts, their primary kinds are determined by their intentional properties which are in turn determined by their producers' creative intentions (Baker, 2007, p. 36). Constitution relation necessarily holds between objects which have different primary kinds. Moreover, constitution brings into existence a new object which has a higher-primary kind than its constituters (Baker, 2007, p. 34). When an artisan makes a table from a hunk, she brings into existence a new object: a table which is constituted by a hunk. The table and the hunk occupy the same space at the same time. However, they are not identical because they differ in their persistence conditions. The table can survive through a replacement of one of its legs, but the hunk cannot. An object and its constituent share properties including their primary kind properties because they have genuine unity and they are spatially coincident (Baker, 2007, p. 37). The primary property of a table is to keep some objects on it, and the table has this property non-derivatively. The hunk has the same property derivatively through constituting the table.

Baker's ideas on primary kinds are reflected by the essential properties of the digital artifacts in Wang et al.'s account. In Wang et al.'s view, when a digital artifact has a different essential property, a different higher-level object comes into

existence. Remembering our program TickEd again, when the programmer of the TickEd decides to run the code base she wrote, she produces a program. The essential property of the program TickEd is its program specification which is determined by the creative intentions of its producers. In Wang et al.'s view, the software artifact cannot survive through losing its essential properties or through some change in its essential properties. A code cannot survive through any syntactic change while its alleged constituted object, the program can keep its identity through these changes (Wang et al., 2004, p. 325). As I explained above, in this part, the constitution relation seems to apply to these two software artifacts: code bases and programs. Although objects are abstract, the case is similar to the hunk and the table in a way in which the code base seems to derivatively become a program by constituting a program. A code and a program can be accepted as different primary kinds, and their constitution relation might hold if we can provide an argument about excluding spatial properties for constitution relation.

The problems regarding the essential properties and their representation of primary kind notion begin with the distinction between a program and a software system. The main issue here is that the distinction between a program specification and a system specification -which are essential properties of a program and a software system respectively- is ambiguous. Moreover, I do not think the differences between a program specification and a software system specification is sufficient to argue that they bring into existence two different (primary) kinds of objects: a program and a software system respectively.

According to Wang et al., the second kind of software artifact, which is created during the programing process, is a software system. A proper function of a

software system is producing a desired change in a data structure only inside a computer (Wang et al., 2014, p. 325). When a programmer determines an internal specification during her programming process, she creates a new kind of entity —a software system— which has a different essential property from both a code base and a program. The essential property of a software system is an internal specification which is different from a program specification. An internal specification concerns the phenomena within the program and the operating systems, without concerning the program behavior whereas a program specification concerns the program behavior as well. The problem is that these two essential properties, a program specification and a software specification, overlap in describing the functions of a software artifact.

The distinction between a program specification and a software system specification is described by Wang et al. as follows:

> 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, abstracting away from the behavior. Note that, in the way we define 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 a program specification and a software system specification overlap in the functional specification (Wang et al., 2014, p. 325).

Although the distinction between a program specification and an internal specification is not clear to me, they argue that they are different essential properties, and each bring into existence a different kind object. As far as I understand, a program's essential property, the program specification, is partly about how the functions are carried by an algorithm. In their formulation, Wang et al. exclude this how part from the internal specification. They think that although a program specification and an internal specification overlap in function specification, which is

the part that determines which functions are carried by a certain software artifact, the way in which these functions carried is extracted from the internal specification. Since the algorithm is a language independent set of instructions, when they exclude how the functions are specifically carried from the system specification; they exclude how specifically the algorithm works.

This distinction might be important for software engineering. However, ontologically speaking, I do not think the distinction is sufficient enough to support their claim on how each of these specifications bring into existence different objects which fall under different primary kinds. If we accept that the primary kind properties correspond to the essential properties in Wang et al.'s view, then since the distinction between the essential properties of these two objects is not clear; then the reason they belong to different primary kinds is not clear either. If they do not belong to different primary kinds, then according to Baker's view, the constitution relation cannot hold between these two objects. We cannot say that the program and the software system have a genuine unity. There is no unified individual -the-software-system-constituted-by-a-program. These two objects overlap in their essential properties, which means they share functional specification as a part of their essential property. A program has certain functions because these functions are specified as a part of its essential property: a program specification. A software system has the exact same function because the exact same functions are specified as a part of its essential property: a software specification. Therefore, these two objects a program and a software system, do not share their properties through a genuine unity, instead they have the same properties because each of them separately has these properties. Even if the program does not constitute a software system at any point of its

existence, it can have the same functions. If it were the same case in the example of a hunk and a table, a hunk would have been a table because a part of its essential property consists of being a table. However, in Baker's view a hunk is a table through constitution and genuine unity. The hunk has the property of being a table derivatively through constituting a table, without constituting a table a hunk cannot have a property of being a table.

Assuming that a program and a software artifact are two different primary kinds, the constitution relation should hold between them in a way in which similar to the illustration below (I will refer to this case as "the case X"): If our programmer changes the code base of the TickEd program, then she merely changes the constituent of the program. We still have the same program TickEd. However, if she changes its algorithm, replacing the Selection Sort algorithm with the Merge algorithm, then we have a different program specification, and therefore, a different program (TickEd-B). According to Baker's constitution relation, the software system is now constituted by the TickEd-B program. Since the constituter of an object can change without changing the identity of the constituted object, we are still dealing with the same software system. The program TickEd and TickEd-B are two different programs which constitute the same software system at different times. While the software system can survive without the program TickEd, it has to be constituted by another program, in our example it is the TickEd-B program. When the programmer replaces the Selection Sort algorithm with the Merge Sort algorithm, she also changes the constituent of the software system: the TickEd program. It is like changing some piece of clay from a statue. Furthermore, in this way, a program derivatively has the essential property of a software system, a system specification,

through constituting a software system. Moreover, a software system shares the essential property of a program through their constitution relation. However, the way Wang et al. argue for the constitution relation between a program and a software system is different from the case X.

According to Wang et al., there are two different cases concerning the relation between a program and a software system:

The first case is similar to the case X. When our programmer follows a linear approach, which means she first writes the code base, then produces the program: TickEd. After she produced TickEd, she replaces its algorithm, and she produces a new program TickEd-B. She thus produces a software system which is constituted by the TickEd-B program. In the case that the programmer follows a linear approach, Baker's notion of constitution holds between the program and the software system in Wang et al.'s view. Under the conditions that Wang et al. can provide an argument for their claim, a program specification and a software system specification are distinct essential properties, and they bring into existence objects which fall into different kinds; the constitution relation prima facie works in the way in which it works in the case X.

However, there is a second case which is extremely different from the case X. If our programmer follows a different approach such that she produces a new software system from an already existing software system, instead of producing it after producing her program TickEd, then the result is different from the previous case. In this case, first, she produces TickEd, with the Selection Sort algorithm again, exactly like our first case, then, different from our first case, she determines an internal specification and creates a software system (let us call this s1) which is

constituted by the TickEd program. After this point, if she decides to replace the algorithm of her software system which is already constituted by the TickEd program, then she creates a new software system (let us call this s2), in addition to a new program. These two software systems, s1 and s2 are constituted by two different programs: TickEd and TickEd-B respectively (Wang et al., 2014, p. 332). There are two different software systems because they are constituted by different programs at the same time. Hence, they have incompatible properties at the same time, and by Leibniz's Law they are different individuals. These two software systems can exist at the same time with, and, independently from, each other (X Wang et al., 2014, p. 332).

The problem is that although we have the exact same change in both of these cases, a replacement of the algorithm, we end up with two different and inconsistent cases. In the first case, we have only one software system. It endures after the replacement of its algorithm. In the second case, we have two different software systems each one constituted by two different programs. The reason for this is that our programmer has adopted the change at different stages of her programming process.

According to Wang et al. in the first case scenario, since the programmer has not created an internal specification before she replaced the algorithm, she has not produced a software system which is constituted by TickEd. Assume that she replaced the algorithm of TickEd at time t, and produced another program TickEd-B. After she produced TickEd-B, she produced a software system by determining an internal specification which is constituted by TickEd-B. The programmer has not produced a software system before time t, because she has not determined the

essential property of a software system before time t. This seems to be their reason to think that after time t, there is only one version available in the first case. One software system which is currently constituted by TickEd-B, and (allegedly) previously by TickEd (Wang et al., 2014, p. 325). However, if this is the case, this claim is odd because, by their description there is no constitution relation between TickEd and the software system. The time the software system is constituted by TickEd cannot be before t, because they think there is no software system before t. The program TickEd cannot constitute the software system after time t, because after time t TickEd-B constitutes the software system. Moreover, they argue that a software system cannot be constituted by two different programs at the same time as I mentioned before. Hence, the relation between these two programs, and under which conditions and how they constitute the same software system remains unexplained in their account. This is the reason their constitution relation does not fit the case I describe in the case X.

Furthermore, if the software system in our first case scenario is constituted by TickEd and TickEd-B at different times as they claim, then the consequence of these two cases contradicts with the transitivity of identity. Let us assume that our programmer works on the TickEd program. She adopts the Selection Sort algorithm, and she replaces the algorithm at time t, again. Any time before t, our software system in the first case scenario is identical to one of the software systems in our second case scenario: s1. These two software systems have the Selection Sort algorithm and hence, constituted by the TickEd program. And after the time t, our software system in the first case scenario is identical with s2 in our second case scenario as both of them have the Merge Sort algorithm, and they are constituted by

the program TickEd-B. The one and the same software system in our first case scenario is identical with the two distinct and different objects, s1 and s2 from our second case scenario, and s1 and s2 are not identical to each other.

Moreover, assume that we have one software system in the first case scenario as they claim, at one time it is constituted by TickEd and at some other time it is constituted by TickEd-B. Furthermore, assume that we have two different software systems as they claim. Since in both cases we are dealing with abstract objects, either she changes the constituent or the essential property of the object; once the software system has been produced we can still have access to both *versions* of the software system at the same time.

In Baker's view, when there is a change in the constituter of an object, the former constituter of the given object is replaced with a new one. Since Baker's notion of constitution requires a genuine unity and spatial coincidence, the same object cannot be constituted by two distinct groups of entities at the same time, at the same place. For example, when an artist revises her work of statue she can change some piece of clay from here and there, but the removed pieces no longer constitute the statue. The removed piece of clay and the statue no longer share their properties with each other.

In the case of Wang et al.'s view, objects are abstract, they do not have spatial coincidence. Hence, there is nothing that excludes the possibility of a software system that can be constituted by two different objects that fall under the same kind at the same time. Under the conditions that Wang et al. argue for, a program and a software system belong to different primary kinds, and a software system is constituted by a program; in both cases we should have the same objects. However,

in the first case scenario, we have the exact same entities that we have in the second case scenario. In the first case scenario we accept them as one and the same entity, and in the second case scenario we have two different entities which are not identical to each other. If Wang et al. were to refuse that a program and a software system belong to two different primary kinds, then they would also have to refuse that their adopted constitution view holds between these objects. As I mentioned, Baker's constitution necessarily holds between objects which belong to different primary kinds. In their argument, there is no plausible explanation for saying that in one case we have one and the same object, and in another case, we have two different objects. Moreover, accepting these consequences leads to an odd metaphysical conclusion as I explained.

The solution is to exclusively argue for either of these: i) there are two different programs, therefore two different software systems in both of these cases, or ii) there is one software system which is constituted by different programs. Since the objects we are dealing with are abstract, we can access the earlier version of the same software system which has a different constituent from our later version, after the replacement.

Assume that we argue for i) there are two different software systems. In this reading, there are two different programs because the producer replaced the algorithm of the program. And there are two different software systems because each of these software systems is constituted by different programs. The program and the software system are either identical with their algorithm, or they are constantly rigidly dependent on their algorithm which means each time the given program and the software system exists, they necessarily have a particular algorithm which they

actually have. In this case, each and every change in their algorithm brings into existence a new program and a new software system. Our programmer can start writing her program, and change the algorithm at different stages of her writing process, then go back to her original algorithm etc. During her producing process she creates various objects, and each time she adopts a new algorithm, there is a new program, and in return a new software system. There is no constitution relation between a program and a software system.

This is the kind of result Wang et al. want to avoid at the beginning of their paper. Their objective is to give an account of digital artifacts which can support some change in their algorithm. They can still argue that the higher-level objects in their account will be constituted by different kinds of objects. Since different software systems can constitute the same higher-level object —a software application—, they can still claim that their account neither identifies digital artifacts with their algorithms nor claims that digital artifacts are rigidly constantly dependent on their algorithm. However, their argument about how these higher-level objects are produced, and individuated is analogous to their argument about program and software system production and individuation. The same problems about identifying a program and a software system after they have undergone certain changes arise for these software artifacts too.

They argue that when the programmer defines an external specification, she produces a new kind of object: a software application. A software application is a software artifact which can be used through a specific machine system which has a purpose to produce an interaction between a user and a computer to produce certain effects in a computer. It is distinguished from a software system and a software

program by its own essential property: an external specification which describes the way in which our software artifact interacts with a user, and determines which kind of machine the software application can be used. After the production of a software application, a programmer can adopt some changes on the external specification of the given software application. The programmer can change the user interface of a software application. A user interface is the part of a software application that a user uses to interact with a computer. Each change in the user interface of a software application brings into existence a new software application because the user interface of a software application is a part of its essential property: an external specification. Moreover, these different software applications allegedly constitute a new kind of entity: a software product, under the conditions that the programmer does not change her choices regarding machine specification, and she determines the high-level requirement of her software product. As I mentioned before, the software product is the closest object to software in our daily usage. The essential property of a software product is its high-level requirements. The high-level requirements of a software product concerns which kind of machine system the software product operates, and which languages the product supports. Hence, software products like Instagram or Facebook that operate in MacOs and Microsoft are different products. Furthermore, if the programmer of these products has not included the possibility to run these products with a system in different languages in their initial requirements, then country-oriented versions of these products are different programs, even if they have the exact same high-level requirements (Wang et al., 2014, p. 329).

The same problems I have investigated through the TickEd example arise again. The time at which the programmer adopts certain changes determines the

identity conditions of the artifact at hand. The circumstances concerning when there is a new kind of artifact, and when there is a replacement in the same artifact are dependent on the producing process of the given product whereby the producers make these adaptations. Furthermore, these two artifacts, a software application and a software product, also overlap in their essential properties, both of these artifacts' essential properties include machine specification which means they both need a specific machine system to run. Moreover, how the constitution relation holds amongst a software system, a software application, and a software product remains unexplained. The explanation they have provided is parallel to their explanation regarding the relation between a program and a software system, as I argued above. And this explanation is problematic for the same reasons. Again, they fail to explain why these objects belong to different kinds. They fail to provide a criterion to individuate any of these objects. Moreover, even if we were to accept that these objects belong to different kinds, and if we accept their way of identifying these objects, their view still gives rise to an odd metaphysical conclusion.

To illustrate, let’s assume that we have a twin earth which is very similar to our world. In our world the initial programmers of Facebook did not include the possibility of running the Facebook product with a Turkish language system, rather they include it later. Assume that in the twin earth, the producers of the Facebook product include the possibility of running their program with a Turkish language system during their initial programing process. In our world, the Facebook product for English and Turkish language systems are two different products. Since the initial requirements of the Facebook product does not include the Turkish language system, adding the Turkish language system in the requirements of the product brings into

existence a new product. The reason for this is that the requirements of a software product is the essential property of the software product. However, in the twin earth the English version of the product is the same product with the Turkish version since both of these language systems are included in the first requirements of the product. We have two different software products in our earth which are not identical to each other, but they are both identical to one and the same product in the twin earth.

There might be an objection that since the essential property of a software product includes being operated by a specific machine system, software products have to be implemented to a system in order to come into existence. However, there is no reason to include the actual execution of the product in its essential property, and further to exclude the possible execution of the product from its essential property.

The idea that ii) there is only one software system which is constituted by two different programs, sounds more plausible. But this idea has not been proven in Wang et al.'s view either.

As I argued throughout this chapter, their argument fails to explain the distinction between different kinds of software artifacts. Under which conditions a different kind object comes into existence is not clear. The main problem with their account is that they distinguish these objects by their essential properties, and some of these allegedly different essential properties overlap (such as a program specification and a software system specification and a software application and a software product). Hence, the way they distinguish these objects remains ambiguous. Furthermore, under which conditions an object constitutes another, and how this

constitution relation holds has not been explained in their view. Moreover, the conclusion Wang et al. arrive at about the individuation of software artifacts violates the transitivity of identity.

Although their choice of constitution notion does not work, another constitution notion would remedy these points. There are some benefits of adopting a constitution relation such as the fact that the constitution view is non-reflexive and asymmetric. Given that in Wang et al.'s account things like algorithms and code bases can exist independently of a higher-level object such as a program and/or a software product and not vice versa, a non-reflexive and asymmetric relation would be beneficial to Wang et al.'s account. The constitution relation allows arguing for different persistence conditions for an object and its constituter. In the following chapter, I will examine another constitution view argued by Simon Jonah Evnine which does not require spatial coincidence, between an object and its constituter.

CHAPTER 4

A TWO-FOLDED CONSTITUTION

Evnine argues that constitution is a "being made out of" relation (Evnine, 2009). This notion of constitution differs from Baker's notion in some fundamental ways. In Baker's view, the constitution relation is about the function of an object which means that the purpose of the production of an object is in locus (Baker, 2004, p. 100). In Evnine's view, constitution is a relation between an object and what the object is made of, and the intentional actions which brought objects into existence. According to Evnine, the identity of artifacts is determined by the acts of these intentional making processes (Evnine, 2009, pp. 213-216). Evnine's notion of constitution does not require spatial coexistence. Moreover, abstract objects such as musical works, fictional characters, and languages are made out of some entities which are intentionally selected by their producers. Hence, they are constituted objects (Evnine, 2009, pp. 213-216). In this chapter I will examine Evnine's notion of constitution and I will argue that his notion of constitution can solve the problems that Wang et al.'s view struggles with.

Evnine argues for a monist account for artifacts. According to Evnine an artifact is a product of making activities by which the artifact is actually brought into existence (Evnine, 2016, p. 87). His account is based on the idea of "origin-as-act" which is the view that in order to determine the identity of an object, the actions which brought a given artifact into existence have to be examined (Evnine, 2016, p. 87). According to Evnine, origin-as-act is necessary and sufficient to determine the identity of artifacts. The identity of a given artifact is essentially determined by acts

through which it is made, instead of the matter which the given artifact is originally made out of (Evnine, 2016, p. 243). The identity of these actions is fundamentally determined by the intentions that lead those activities (Evnine, 2016, p. 245). For example, let's say I am making a puppet from a piece of cloth. I am sewing up a piece of cloth to give it the shape of a puppet. At the end, I make a puppet out of the given piece of cloth as a result of my making out of activities. If my making out of activities would not have occurred, there would not have been the given puppet. Therefore, in Evnine's argument, the existence of artifacts depends on the existence of intentional human activities, in a way that if it were not for these intentional making activities, artifacts would not exist at all. Moreover, the identity or the nature of artifacts also depends on their producers producing activities which brought that artifact into existence. These claims are parallel to Thomasson's claim. Evnine argues that his view differs from Thomasson on some important points. I will use only one of these distinctions. As I mentioned above, Thomasson argues that one does not have to commit to "magical modes of creation" for artifacts, that is some artifacts such as fictional characters can come into existence by the mere thinking and speaking activities of their producers (Thomasson, 2009, p. 197). She suggests that instead of committing to magical modes of creations we can provide a methodology to determine the identity conditions of objects such as fictional characters. On the other hand, Evnine argues that whether an object can come into existence by the mere thinking and speaking activities of their producers depends on the kind of object and the kind of labour specifically required for the given kind of object (Evnine, 2016, p. 117). I will explain this issue later in this chapter.

For now, Evnine thinks that committing ourselves to "magical modes of creation" is not problematic for certain objects such as abstract artifacts (Evnine, 2016, pp. 116-117).

According to Evnine, when I made a puppet, the exact same puppet could not have been made by another person, because my intentions and actions belong essentially to me. The reason for this is that the identity of actions is determined by the intentions which lead to those actions. The intentions are mental states and they essentially belong to the agents who actually have these certain intentions (Evnine, 2016, p. 245). The important point is that he does not argue that actions are rigidly dependent on their producers which means that whenever there is a certain action it is necessarily produced by whom it was actually produced. The result is the same, but, he thinks that two different individuals cannot perform the exact same actions because two different individuals cannot have the exact same mental states and the exact same intentions. If it were possible for two different people to have the exact same mental states and the exact same intentions, then these two people could have produced the same actions (Evnine, 2016, p. 245).

Moreover, he thinks that someone cannot have numerically identical intentions. Therefore, when I make another puppet which looks exactly like my former puppet, these two puppets are different objects. Since my intentions to make these puppets and my making out of actions of them are different, they are different objects. Hence, in Evnine's account both intentions and actions belong to certain agents whomever they actually belong to (Evnine, 2016, p. 246). Furthermore, the modal properties of the given artifact are also dependent on these making out of actions rather than a general ontological theory about the kind which the given object

belongs to or the matter of the work (Evnine, 2016, p. 139). Although what kind of making out of activities are allowed partially depends on the general ontological theory about the given work, more specifically what kind of labour is required by the kind the given work belongs, the identity of the work is still determined by the identity of the making out of activities which brought the given object into existence (Evnine, 2016). For example, the identity of my puppet is determined by my making out of activities which brought into existence the given puppet. In order to determine the identity criteria of my puppet, or the modal properties of my puppet, my making out of activities regarding the given puppet should be examined rather than the matter of my puppet or a general ontological theory about artifacts. This is the reason he calls his view as an origin-as-act view. If we were to accept my puppets as artworks, then making out of intentions and activities are partially dependent on, or being more specific, they are restricted by some general ontological theory about the work of art. I will explain this part in the sub-chapter on making out of activities.

Although actions and making out of activities have some differences, the identity of artifacts are determined by creative actions which bring them into existence. For this reason, the identity of actions should be examined to understand the account. For this reason, first, I will explain Evnine's account on action identification to introduce some preliminaries to his account on artifacts (4.1). Second, I will examine his account on actions of making to explain how the constitution relation works in his account (4.2). Third, I will examine his account on abstract objects and their application to digital artifacts (4.3). I think Evnine's notion of constitution solves the problems that Wang et al.'s view faces. Finally, I will discuss an additional issue about revising abstract artifacts.

4.1 Actions and artifacts in Evnine's account

According to Evnine, actions are similar to artifacts with regards to the way in which they are both made (performed by) by some agents, and they are made with some intentions. He makes a distinction amongst basic actions, lower-level actions, and higher-level actions.

Basic actions are performed by making some bodily movements like raising a hand, moving a finger, etc. Bodily movements are different from actions because they do not have an agent as their components, they have body parts as their subjects (Evnie, 2016, p. 231). When I intend to raise my hand and do so by raising my hand, I make a basic action which is made by my bodily movement: my hand rise. If I fail to raise my hand for some reason, such as having lost some brain tissue which narrowed my physical abilities, then I fail to perform a basic act too. The lower-level actions are made by basic actions, like a click on "enter" on my keyboard. My clicking on the keyboard is a lower level action, which includes a basic action which is moving my finger, and the basic action of my finger moving is made of my bodily movement: my finger decreases vertically. I could have used my elbow to click on enter, in this case my lower-level action would have been made by another basic action, moving my elbow, and this basic action would have been made by my elbow movement as its bodily matter.

The higher-level actions are made by basic actions or from some lower-level actions. One example would be, singing a song. My singing a song is a higher-level action which is made out of my basic actions such as opening my mouth, moving my tongue and my lips, and some lower-level actions like uttering the lyrics of a song

hopefully melodically, producing some noises. Now, if I fail to sing a song, I still perform an action like uttering some lyrics and making some noises. Although I can fail on my intention to sing a song, assuming that bodily movements and my basic actions are still there, I can still perform my lower-level actions like uttering some lyrics and/or making some noises. Even if I fail in those actions too, unless there is a neurological or physiological problem with my body which makes it impossible for me to make some basic actions from some bodily movements, I still succeed on my basic actions like opening my mouth or moving my tongue. Moreover, I can perform a higher-level action from basic action such as dancing. In the case of dancing, I make some basic actions out of my bodily movements, then I make a higher-level action which is made out of my various basic actions.

The identity of my action is dependent on my intentions, and the description of my action. An action description contains the agent of an action, the time of an action, and its action type. According to Evnine, two actions are identical under the following identification criterion:

> Act-Ind) An action A1 is identical to an action A2 iff A1 has a canonical action description D1, A2 has a canonical action description D2, and either i) the agent, time, and action type D1 are identical, respectively, to the agent, time, and action type of D2, or ii) it is correct to say that the agent of A1 performs the action of the type specified in D1 (at the time specified in D1) in performing the action of the type specified in D1 (at the time specified in D1) *in*[2] performing the action of the type specified in D2 (at the time specified in D2) (Evnine, 2016, p. 236).

To illustrate, let's assume that I am singing a song at 1 p.m. The action description (D1) contains the action type which is singing a song, the agent of this action which is me, and the time of my action which is 1 p.m. There are two ways for another action to be identical to my singing. The first one is that this other action has to have

the same agent (me), the same action description (D1), and the same time (1p.m) with my singing. Second one is that I act in the same way described in D1, at 1p.m (the time specified in D1) *in* doing another action, for example, annoying my sister which has another action description (D2), and I annoy my sister at time specified in D2. The word "in" emphasizes here that I do not have an intention to annoy my sister. I might be even unaware of her annoyance. However, I still annoy my sister in singing a song.

On the other hand, assume that I have an intention to annoy my sister for some reason, and I sing a song to fulfill my intention to annoy my sister; in this case, my singing a song and my annoying my sister are two distinct acts. The reason for this is that both of these actions are led by distinct intentions. In this case, I annoy my sister *by*[3] singing a song. My act of annoying my sister is done by my singing in both cases. However, in the case that I want to annoy my sister, there are two intentions; to sing a song, and to annoy my sister. My two actions; singing a song and annoying my sister are dependent on two different intentions respectively to sing a song, and to annoy my sister. They are two different action types, and their action descriptions are different. I do not perform either of them *in* performing another, instead I do one action annoying my sister *by* doing another action singing a song with a nested intention to do both of these actions. If I were intent on annoying my sister yet I failed to do so, I would still perform an action of singing a song (Evnine, 2016, pp. 221-225).

---

[2] The word "in" here emphasizes unintentionally performing an action.
[3] The word "by" here emphasizes intentionally performing an action in order to perform another action.

Furthermore, the modality of my action —annoying my sister— also is not dependent on my singing. I could have annoyed my sister by loudly tapping on my table, clapping my hands, etc. (Evnine, 2016, p. 247). In the case that I annoy my sister by singing a song, one of my actions —annoying my sister— is made out of another action, namely, my action of singing. It could have been made out of another action. It could have occurred at another time, or my sister could have been annoyed a few minutes earlier or later etc. The reason for this is that my annoying action is not a basic action. My action of annoying someone is a higher-level action which is constituted by some other actions, and it could have been constituted by different actions. When someone fails to perform a basic action, she fails to act, but when someone fails to perform a higher-level action, she performs its matter —either a basic action or a lower-level action— regardless of whether she fails at, or does not perform, the higher-level action (Evnine, 2016, pp. 225-229).

According to Evnine, both artifacts and actions have plural matter. In the case of actions, the matter of a higher-level action is produced by their agents such as my typing actions being produced by basic actions, (e.g. moving my fingers). In return, basic actions like moving my fingers are produced by certain movements of my fingers. The actions and the artifact objects differ from each other with regards to their production processes. The artifacts and actions are necessarily made by some agents. They are both raised from intentions. However, performing an action and making an artifact are two different processes. In the case of artifacts, agents need actions of making because they use matter which already exist and make something else out of this matter. For example, a carpenter makes a table by using a hunk or some other material which already exists independent of himself. In the case of

actions, the matter of the actions is also produced by their agents (Evnine, 2016, p. 232). As I illustrated in the example of annoying my sister, I make my annoying action out of a lower-lever action: singing a song. I also could have made it from a bodily movement. The fundamental point is that I make my action of annoying someone from other actions which exist because I made them. I produce the matter of my action of annoying someone with my singing actions. In the case of basic actions, an agent makes her action out of bodily movements. Hence, under the circumstances that an agent makes a higher-level action from a lower-level action or a bodily movement, she produces the matter of her higher-level action by performing the lower-level action or bodily movements with the right intentions. Since the matter of the actions is produced by their agents, there is no requirement for acts of making for actions. The agent simply produces an action from another action or from bodily movements with the right intentions (Evnine, 2016, p. 232).

In the case of making artifactual objects, actions of making are required since acting with certain intentions does not produce the matter of artifactual objects. The tree which our carpenter cuts down to make a table would not have been there without its seeds, however it could have grown without our carpenters' intentions under different circumstances. The matter of the table has not been produced merely by our carpenter's intentional actions, instead it has been produced from a tree which was grown from a seed. Our carpenter's actions involved in the process of growing the tree. However, the tree could have existed without our carpenter's intentional activities. The seed of our tree could have grown naturally without any human activities involved in the process. However, the table is necessarily made by an agent; without certain intentional activities of an agent there would not been a table.

Hence, even in the case that agricultural process might be involved in our table making activities, the matter of the table has not been produced entirely by our agent's activities.

In the case of actions, there is an agent, her intentions lead the basic actions, and she makes lower-level actions or higher-level actions from these basic actions without requiring additional actions of making. These basic actions could not have existed without the intentions of their agents (Evnine, 2016, p. 244). I could have some involuntary tics, or spasms, but without my intentions these bodily movements would not be actions. Since actions require an agent and bodily movements have body parts as their subjects, my tics and spasms would be considered bodily movements, but they would not be considered basic actions. Therefore, in the case of artifacts, there should be some action of making and those actions should involve both some kind of work or labor on a matter, as well as creative intentions of an agent (Evnine, 2016, p. 244).

Another distinction between actions and artifacts is that artifacts can survive through some changes but actions cannot survive through changes. Although actions have some modal flexibility regarding their matter, they cannot change after they are performed (Evnine, 2016, p. 246). Evnine thinks that the identity of actions is determined by the intentions which lead the actions. In order to determine whether a counterfactual action is the same action with an actual one, we should check whether these counterfactual intentions which lead these counterfactual actions are the same as or different from the actual intention (Evnine, 2016, p. 246). In the example of my singing, while I could have sung another song at 1 p.m, after I already sang a song at 1 p.m, I cannot change that specific singing action. Since any alleged change in my

already existent action would have been produced by a different intention, and in turn would have a different action description; it would be another action (Evnine, 2016, p. 236). This is the reason Evnine argues that, within the boundaries of three-dimensionalism, actions cannot change their matters after they are produced (Evnine, 2016, p. 237). If we were to accept four-dimensionalism instead of three dimensionalism, then the actions would be considered objects and they can survive through change. However, Evnine argues within the boundaries of three-dimensionalism. I also remain within the boundaries of three- dimensionalism in this thesis.

4.2 Making out of actions: a two-folded constitution theory

I have explained that in Evnine's view, artifacts are products of the actions of making them, and constitution is a "being made out of" relation. The phrase refers to the intentional productive actions of an agent regarding producing a certain artifact. Evnine suggests a two folded theory of being made out of. He thinks that there are two denotations of the phrase: "being made$_E$ out of", and "made$_S$ out of" (Evnine, 2009, p. 210). The first denotation is about the actual making process, and it refers to the events through which a producer makes an artifact out of something. This version is called "made$_E$ out of" relation (Evnine, 2009, p. 210). The made$_E$ out of relation is a three-place relation amongst an agent, the objects or entities which the constituted object is made of, and the object itself (Evnine, 2016, p. 74). When I make a puppet out of a piece of cloth by stitching the cloth, I make$_E$ a puppet out of a piece of cloth. My activities of stitching the piece of cloth is described by the phrase "making$_E$ out of". The second denotation of the phrase is about the enduring

relation of an object, and it holds between an object and what the object is made out of. This version of the phrase is called "made$_S$ out of" relation (Evnine, 2009, pp. 211-210). The made$_S$ out of relation is a two-place relation[4]. Since the constitution relation is about the persistence and endurance conditions of objects, the emphasis is on the made$_S$ out of sense of the phrase. For example, when I am making a puppet, after I finished my stitching actions; my puppet is made$_S$ out of the piece of cloth which I used to make the puppet. The two-folded constitution notion aims to provide a temporally and modally flexible account of constitution by formulating under which conditions an artifact comes into existence via the made$_E$ out of sense of the phrase, and under which conditions an artifact continues to exist via the made$_S$ out of sense of the phrase. Through these two senses a sufficient condition for constitution is set up. Evnine's formulation of his two folded theory is as follows:

> MADE *y* is made$_S$ out of *x* if and only if some agent made$_E$ *y* out of *x* and *x* has not been replaced, or some agent made$_E$ *y* out of *w*, and *x* is related by the ancestral of replacement to *w* (Evnine 2009, 212).

The formulation emphasizes the making actions of an agent, instead of the object itself or the matter of the object. The same making out of actions necessarily bring into existence the same object, regardless of whether the matter of the object has been replaced or not. For instance, I can start stitching my puppet out of a piece of cloth (let us call that cloth-A), then for some reasons I can change the piece of cloth that I used in the first place, and then use another piece of cloth (let us call that cloth-B). In this case, I am still working on the same puppet. First, I made$_E$ my puppet out

[4] Evnine thinks that both made$_E$ out of and made$_S$ out of relations have a time component, however he often intentionally ignores this component because he thinks these objects are temporally flexible (Evnine 2016, 74).

of the cloth-A, at this point my puppet is made$_S$ out of the cloth-A. When I replaced the cloth-A with cloth-B, the puppet is made$_S$ out of the cloth-B. There is only one puppet after my making out of activities. Although I made$_E$ my puppet out of the cloth-A at first, my puppet is made$_S$ out of the cloth-B. Since the cloth-B replaced the cloth-A, the puppet is made$_S$ out of the cloth-B. The cloth-B constitutes the puppet by replacing the cloth-A.

The constitution relation holds between an object and its constituent by which the object is made$_S$ out of (Evnine, 2009, p. 212). Evnine's description for the constitution of an artifact is the following: if an object is an artifact, the constituent of that object constitutes the object if and only if the object is made$_S$ out of the given constituent (Evnine, 2009, p. 212). Hence, if the puppet is an artifact, the cloth-B constitutes the puppet if and only if the puppet is made$_S$ out of cloth-B.

Furthermore, an agent's actions of making out of something necessarily bring into existence that intended object. When the intentions of the agent have not been satisfied, there is no object at all. There is something called *scrap*
[5] which describes the mess the agent left behind. For example; when I am making a puppet, I work on some pieces of cloths. If my making actions are not successful, then I also fail to bring something new into existence. The piece of cloth I am working on does not look like a puppet at all. In this case, the piece of cloth in my hands is *scrap*. If I had been successful, I would have made a distinct object, a puppet, instead of scrap (Envine, 2016, p. 128). Hence, if I fail to make a puppet, I

also fail to bring into existence a new object. The reason for this is that artifacts are necessarily the result of some agents' making out of activities; if such activities have not been performed, there is no artifact. These making out of activities are determined by the specific intentions of a maker with regards to producing a certain object. If I decide to make a puppet; under the condition that I succeed on my making activities, then the end work of my activities regarding this puppet making process is necessarily a puppet. If I fail to make a puppet, then I produce scrap which is not an object. In the case that I fail, after my failed attempt, I can use the leftover piece of cloth as something else such as a cover for an object. However, my mere using-as activities are not sufficient to bring into existence a new object. At this point, Evnine makes a distinction between ready-made objects and using-as cases (Evnine, 2016, pp. 133-135). I will explain this distinction later in this sub-chapter.

Furthermore, the modal properties of an artifact are also determined by the making out of actions which brought that artifact into existence. If I were to use another piece of cloth or another material, (a piece of paper, for example) my puppet would have been the same object as long as my intentions leading to my making out of activities are consistent with these alternations and my activities of making out of the puppet are the same. Similarly, I could have used the material of my puppet to make another object like a sock. Moreover, after I replace cloth-A with cloth-B on the previous example, I can make another object with cloth-A. I could have made another puppet too. My making out of actions are different from each other because

[5] The term "scrap" belongs to Risto Hilpinen: "Proper authorship requires that the character of the object produced should fit the author's intentions (to some degree) and not merely depend on them. If an author fails in every respect, he does not produce a genuine artifact, but only "scrap"; he is not an author of anything in the "intentional" sense of the word (1993, 160-1 as cited in Evnine 2016, p. 239)."

their action descriptions are different from each other. Puppets I produced as a result of these actions would have been different too (Evnine, 2016, p. 245). The identity of my making out of actions are determined by my intentions which lead to my actions and the context of my actions. The identity of intentions is parallel to the identity of actions which I explained in the previous chapter. Intentions are mental states, and they are produced without making out of activities. They necessarily belong to particular agents whomever they actually belong.

The identity of artifacts is dependent on their making out of activities and at some level, the content of the given artifact. The content of the making out of objects are dependent on mostly the object's kind because the intentions lead to certain making out of activities mostly tied to which kind of object is produced at the end of these activities (Evnine, 2016, p. 117). There are two meanings of this content dependence. The first one is that, the kind our object falls under determines some kind-specific features of the given object. For example, if I adopt a hobby of making puppets, I assume I will progress through a process and my first work will be aesthetically and qualitatively different from my five hundredth work. If I were to make my five hundredth work as a first one, it would have been more impressive and would have had different features than it actually has in terms of my puppet repertoire. These differences might not be essential for my hobby work, however, for artworks they are included in the content of the work (Evnine, 2009, p. 139).

The second one is that the conditions under which a given object can be brought into existence is dependent on the object kind into which the given object falls (Evnine, 2016, p. 117). For example, Evnine thinks that a sculpture can come into existence by choosing an object as a sculpture under the condition that we accept

it is possible for an artwork to be created in this way. At this point, Evnine draws a difference between ready-made objects and using an object as something else; this distinction is dependent on the content dependency of making out of activities (Evnine, 2016, pp. 133-135). For example, I could use a bed as a sofa, and only as a sofa, and never use it as a bed. My usage would not turn the bed into a sofa. The bed would remain as a bed which is used as a sofa. The reason for this is that being a sofa is a substantial kind instead of a phasal kind, and a bed is not necessarily a sofa (Evnine, 2016, p. 134). The difference between a substantial kind and phasal kind is about what a kind essentially is. For example, human is a substantial kind whereas, a child and adult are phasal kinds. While a human being is growing up, she becomes a child, a teenager, an adult but; these phasal kinds do not bring into existence a new thing which is distinct from the human (Evnine, 2016, p. 7). In Evnine's account artifacts are substantial kinds. In the case of making a bed or making a sofa, there is something new added to the world of things. However, in the case of using a sofa as a bed there is nothing new added to the word. Although there are kinds of sofas which can be turn into a bed when some parts of it are rearranged, a bed is not necessarily a sofa. Sofa making activities are different from using something as a sofa. Again, the making out of activities of an artifact requires some labor which carry the intentions of the maker to the object (Evnine, 2016, p. 117). If it were the case that simply using something as a sofa is sufficient to make a sofa, then my bed would have been a sofa.

In the case of works of ready-made works of art such as Duchamp's Fountain, the topic is partly dependent on whether an artwork can be made by an already existing object or not. In the case of Duchamp's Fountain, assuming that an object

can be made by an already existing object, then the urinal constitutes Duchamp's Fountain (Evnine, 2016, p. 135). However, the situation might be similar to the case of using a bed as a sofa. If we were to argue that Duchamp uses the urinal as a sculpture, instead of creating a new object, a sculpture, then the case would be the same with the bed and the sofa example. According to Evnine, there is no basis to know which one is the case when it comes to artworks. Under the conditions that an artifact can have another artifact as its only matter, and we can produce artworks or things in general by thought or talk alone, then ready-made objects like Duchamp's Fountain, are new objects (Evnine, 2016, p. 135). If one were to refuse these conditions, then ready-made objects would not be new objects; instead they would be some objects which are used as other objects (Evnine, 2016, p. 135).

Evnine suggests that in the case of artworks, accepting that mere talks and thoughts can bring artworks into existence, the idea that an artwork might be constituted by an already existing object is not problematic. The reason for this is that creative actions and making out of activities involved in the case of artworks are different from the ones involved in the case of ordinary objects (Evnine, 2009, pp. 214-215; Evnine, 2016, p. 135). According to Evnine, in the case of making artworks, the creative intentions of the artist can be carried by mere thoughts and talks. The artist can realize her intentions when she see the object which corresponds to her intentions. Hence, the case is different from using something as another object. Moreover, he thinks that even the possibility that something can be constituted merely by another object, as well as the possibility that some objects might be brought into existence by mere talks and thoughts are sufficient to argue for ready-made objects (Evnine, 2016, p. 135). His ideas on abstract artifacts are partially

supported by his acceptance of the possibility of ready-made objects. In the following sub-chapter, I will examine his account on abstract artifacts, and I will attempt to apply his account to digital artifacts.

4.3 The two-folded constitution view applied to digital artifacts

In Evnine's account, abstract objects such as musical works, fictional characters, and languages are also artifacts which are produced by their producers, and they are constituted by some entities which are intentionally selected by their producers (Evnine, 2016, p. 136). Evnine thinks that when a composer composes a work, she works on a sound structure with certain kinds of intentions in her mind to reach at a certain kind of work. Under the conditions that there is a kind which is a musical work, and the sound structure becomes a musical work through the intentional work of its composer(s)', the musical work is made$_S$ out of that sound structure through its composers' making$_E$ out of actions of a musical work. The end product is the musical work which is made$_S$ out of the sound structure (Evnine, 2009, p. 214). Since digital artifacts are also abstract artifacts, Evnine's formulation should be applicable to them too.

If we apply Evnine's account mutatis mutandis to Wang et al.'s argument on software artifacts, we have the following case: The programmer starts by choosing an algorithm, then writes a code base to implement that algorithm. By writing the code base, she makes$_E$ a program out of a code base. Given that there is some kind which is a program, and the code base becomes a program through its producer's intentional creative activities to make a program out of the given code base, then the

program is made$_S$ out of the code base. That the program is made$_S$ out of the code base means that the program is constituted by the given code base.

However, I think an algorithm is more suitable as a main constituent of the program. In the case of musical works, Evnine argues that they are constituted by their sound structures. In the case of digital artifacts, algorithms play a role similar to the role of sound structures in musical works. An algorithm is an abstract entity which is language-independent, and it is a pattern of instructions (Wang et al., 2014, p. 323). In the ontology of musical works, a sound structure is also an abstract entity, like numbers and sets (Levinson, 1980; 2001; Evnine, 2009; 2016). Moreover, in the case of digital artifacts, an algorithm describes which steps will be taken and how these steps will be sequenced. Similarly, in the case of musical works a sound structure provides the sequence of instructions for a musical work which describes how a musical work will be performed. For this reason, I will accept in the following that digital artifacts are mainly (if not only) made out of an algorithm. We can also accept that a digital artifact has plural matter and some other entities such as an algorithm or, a code base are its constituents. Since Evnine argues that both actions and artifacts can have plural matter, Evnine's account would still support this claim.

As I argued throughout chapter 3, one of the main problems that Wang et al. face with is to provide a plausible explanation about how the constitution relation holds between their objects. The problem is mostly caused by their adopted notion of constitution which is not applicable to their account. Their adopted notion of constitution requires spatial coincidence, and necessarily holds between different kinds of objects. However, they argue for abstract objects which does not have spatial properties. Moreover, they do not provide a plausible argument about their

objects belonging to different kinds. In Evnine's account there is no requirement of spatial coincidence between an object and its constituter. Moreover, Evnine's notion of constitution relation does not require different kinds of objects. The original creative activities of the given object are in locus. For instance, when our programmer decides to produce the ticket sorting program I mentioned above, she makes some choices and performs some activities. The end product of her activities concerning this programming process is the TickEd program. The TickEd program is made out of its algorithm through the making out of activities of its programmer. The identity of our program TickEd is existentially dependent on the making activities which brought TickEd into existence which means without such activities there would not be the program TickEd.

The first point that needs clarification is that the making out of activities of abstract objects seem different than the making out of activities of daily objects or concrete artworks. For example, when I make a puppet, I work on an existing piece of cloth. I modify an already existing entity. I can also produce the piece of cloth from a ball of yarn, or some other entities, but there is an existing material which I work on. In the case of musical works and digital artifacts, and other abstract objects, the work of their producers on the constituent matter is not as clear as it is in the case of making a puppet because the constituent matter of a musical work or a digital artifact is abstract. The idea of interacting with an abstract object and modifying an abstract object through these interactions sounds odd. The claim needs a further argument on how it is possible to interact with an abstract object. Evnine thinks that this is not a problem at all. He argues that a composer can discover a sound structure by distinguishing the specific sound structure from other sound structures. The

process is similar to Platonism about musical works, which argues that musical works are sound structures which are discovered. According to Evnine, the discovery of a sound structure is the making out of activities of a composer. The idea is that a composer discovers a musical work by distinguishing it from other sound structures, by eliminating both different and similar sound structures with an intention to reach a certain sound structure she has on her mind. The sound structure which is discovered through this process constitutes the musical work (Evnine, 2009, p. 215). This process of elimination and recognition (of sound structures that fit the intentions of a composer) is the composing labor, and it is the making out of activities of a composer (Evnine, 2009, p. 215). During this process, a composer made$_E$ a musical work out of a sound structure. Once the process has been completed, the composer made$_S$ a musical work out of the sound structure which she has discovered.

As an alternative, a composer's work on the sound structure might also be considered similar to the work of a sculptor who uses a found object as an artwork like in the case of Duchamp's Fountain. A composer might select a sound structure maybe displaying one, or writing its score (Evnine, 2009, p. 215; Evnine, 2016, p. 137). In this case, a composer might discover her own intentions when she finds the corresponding sound structure. Hence, even if the idea that a composer works on an abstract object is rejected, the view still allows making musical works out of sound structures (Evnine, 2009, p. 215). In both of these cases of discovery and using a found object, the artist does not make changes on an abstract object. Instead of this, she selects some sound structures, displays them, and makes some choices to use a sound structure to compose a work. Evnine thinks that these activities of a composer are her making out of activities. The musical work is the end product of these

activities and the identity of the work is determined by its composers' making out of activities (Evnine, 2009, pp. 214-215). As I mentioned in chapter 3, Wang et al.'s view needs to rely on similar assumptions about the creativity of what they define as essential properties of software artifacts. Since defining and determining essential properties of software artifacts imply the same abstract interaction, they also need to rely on concepts such as using ready-made objects, or discovering an abstract object, or creating abstract objects.

As I explained earlier, in Wang et al.'s terminology, a software product is the closest object to our daily use of software. When a programmer decides to produce a software product, the programmer aims to produce some effects in the world through a computer system. She decides what kind of functions her product will carry out and which results her product will produce. She decides on certain features her software product can have. These decisions and choices all together are concerned about defining the requirements of her software product. For example, when the programmer of Instagram —which is a social media platform that allows users to edit and share photos and short videos— decides to produce Instagram, he has in his mind certain functions that Instagram should carry, and certain features Instagram should have. In the case of Instagram, these functions include the editing and filtering of photos and videos, the sharing and viewing of these photos and videos on a profile, as well as some interaction between different profiles such as liking others' photos or videos, etc.

According to Wang et al.'s view, the programmer defines the requirements of Instagram through implementing his choices and decisions about the functions and features of Instagram (Wang et al., 2012, p. 328). In order to implement his decisions

and create the software product, the programmer has to produce certain activities such as choosing an algorithm to implement; writing a code to implement the algorithm; defining a program specification, an internal specification, and an external specification; defining the requirements in a way in which they can be run by a machine system. In Wang et al.'s account, as I argued in detail through this thesis, each of these activities allegedly produces different software artifacts, and each of these artifacts belongs to a different kind. Furthermore, during the producing process, a producer also produces several artifacts which belong to the same kind. The final product is supposed to be constituted by some of these intermediary objects.

According to Evnine, someone can make one action out of plural other actions (Evnine, 2016, p. 221). A programmer can make an action of producing a software product out of a plurality of actions such as choosing the algorithm to implement, writing the code-base, defining the requirements etc. The reason that all these actions constitute a single making out of action is that there is one overarching action which is driven by an overarching intention: producing a certain digital object. In the case of producing a software product, the overarching action is made by all these intermediary actions of the programmer such as choosing the algorithm, writing the code, deciding the requirements etc. Since the programmer has an overarching intention which leads her to perform all these sequences of acts, these intermediary actions are the matter of one single overarching action (Evnine, 2016, p. 221). The programmer intends to produce a software product and plans which actions she will perform in order to produce such an artifact. She deliberately and intentionally follows these intermediary actions to complete an overarching act.

Considering again the previously mentioned ticket selection program, TickEd, the programmer keeps the same intention, which is to produce a digital artifact to pick the cheapest theater ticket. She chooses the same algorithm -the Selection Sort algorithm- to implement her program. After she tests the program she decides to make some changes for developmental purposes, and she changes the code of the program. Assume that our programmer starts with a code base: c1. She makes some changes on this code base for maintanance purposes or writing the code with a different computer language etc., and she gets another code base c2. Her program is now made$_S$ out of c2. She still works on the same program because c2 is the descendent of c1, and c2 starts to constitute the program by replacing c1. I would like to remind that in Wang et al.'s view a program also can survive through the same change.

However, assuming that our programmer still was not satisfied with the results, and she decides to use the Merge Sort algorithm instead of the Selection Sort algorithm. According to X. Want et al., as I mentioned in the earlier chapter, she creates a new program through this replacement. According to Evnine's account, she does not create a new program. As long as her initial intentions concerning creating TickEd is consistent with her act of replacing the algorithm of TickEd, she still works on the same program. The constituent matter of the program changes after her replacement, but, the program can survive through this change. The program TickEd is initially made$_E$ out of the Selection Sort algorithm, and made$_S$ out of (constituted by) Selection Sort algorithm by its producer. Now, the producer replaces the Selection Sort algorithm with the Merge Sort algorithm. Since the Merge Sort algorithm is related to the Selection Sort by the ancestry of replacement. TickEd is

made$_S$ out of the Merge Sort algorithm. Thus, the TickEd program is made$_S$ out of the Merge Sort algorithm. Now, after the replacement of the algorithm, our program TickEd is constituted by the Merge Sort algorithm.

In Wang et al.'s view, as I explained in the previous chapter adopting different algorithms at different times during the programming process leads to contradictory cases. When the programmer replaces the algorithm of a program before she makes a higher-level object, a software system which constitutes the program, she creates a new program, and this new program constitutes the software system. The first problem is that even though when and how the former program constituted the software system was not explained, Wang et al. argue these two programs constitute the same software system (Wang et al., 2014, p. 323). On the other hand, if our programmer first produces the software system, then adopts a new algorithm she creates a new software system too. Moreover, the consequence of these cases violates the law of transitivity of identity. In Evnine's view we do not have the same problem. Regardless of when the programmer replaces the algorithm of a software artifact, as long as she is performing the same making out of activities, she is considered to be working on the same digital artifact.

Moreover, Wang et al. argue that a software product, which is the closest object in their ontology to our usage of software, is constituted by various objects which belong to different kinds. I argued that they fail to explain how these objects belong to different kinds. However, even if we accept Wang et al.'s argument that a software product is constituted by different objects each belongs to different kinds, in Evnine's view there is still one end product which is constituted by plural other objects.

According to Evnine's notion of constitution, assume that we accept that a program and a software system fall under different kinds. Furthermore, a producer can produce one artifact —a software system— by producing another, a program. As I explained in the action part of the discussion, I can perform an action, an action of annoying someone by doing another action, singing a song. Similarly, a programmer can produce a software system by producing a program. The reason for this is that the identity of a program and a software system is determined by the making out of activities of these programs. If making out of actions which bring into existence a program requires different intentions and making out of actions from those which bring a software system into existence, then these two objects are different objects. As we can remember, Wang et al. argue that a programmer defines a program specification by choosing an algorithm, writing a code in order to implement this algorithm, and determining a data structure (Wang et al., 2014, p. 324). In order to produce a software system, a programmer should define an internal specification which requires producing all these actions, and then excluding how the given program behaves in a computer. Wang et al. think that a program specification and an internal specification overlap in their functional specification, however, a program specification contains a program behavior whereas an internal specification does not. Under the conditions that the distinction between a program specification and an internal specification requires distinct making out of activities for each of these objects—for a program and a software system—, and our programmer makes a software system out of a program, then the software system is made$_E$ out of the program. Given that there is no more replacement after this point, then our software system is made$_S$ out of (constituted by) a program. For instance, if our programmer

produces TickEd, and then makes$_E$ a software system out of TickEd program, then the given software system is made$_S$ out of (constituted by) the program TickEd. Moreover, under the condition that our programmer makes a higher-level object— a software application— out of the software system which is produced by TickEd, then the given software application is made$_S$ out of (constituted by) a software system which in turn made$_S$ out of (constituted by) TickEd. In this case, our programmer makes one object out of the other, a software system out of the program TickEd. She uses this new object —the software system— to make another higher-level object—a software application—. Moreover, the software system is made$_S$ out of (constituted by) the program TickEd. Hence, even if we accept Wang et al.'s claim about during the producing process of a software product a programmer produces various objects which belong to different kinds, Evnine's argument gives an explanation as to how the constitution relation holds amongst these objects. I think this is the case Wang et al. aims to reach because they claim that software artifacts are dependent each other by layered constitution relations which means that a higher-level object is constituted by a lower-level object up until the point that a programmer produces a software product.

On the other hand, even if we accept that some of these objects such as a program and a software system are produced independently from each other; and there is no constitution relation between these two objects, we can still claim that these two objects constitute a higher-level object. For example, under the conditions that i) a software system and a program require distinct making out of activities in order to come into existence, ii) and a software system can be produced without being made out of (constituted by) a program; then our programmer can use these

two distinct objects to make another higher-level object, a software application. This higher-level object would be made out of plural objects, a program and a software which means this software application would be constituted by the given program and the given software system.

As long as we can accept that a software product is constituted by other software artifacts, whether these other artifacts have a layered constitution relation or not, the given software product is made out of (constituted by) these other software artifacts such as a program, a software system, a software application etc. Until the production of a software product, all these software artifacts are either: a) produced with nested intentions and related to each other by layered constitution relations, and at the end they constitute a software product; or b) produced independently from each other and together they constitute a software product as plural constituents of the given software product. In each of these cases, the software product is brought into existence as a result of one overarching making out of action which constitutes various intermediary actions such as making out of different software artifact and/or making out of a software product from different software artifacts.

As a brief summary, in Evnine's view we can accept that a software product is made out of several software artifacts starting from an eternal abstract object: an algorithm, and includes artifacts such as a code base, a program, a software system etc. We can either argue that i) these software artifacts are related to each other by layered constitution relations, and hence each of them is an intermediary object within the process of producing a software product ii) or these software artifacts are not related to one another with a constitution relation but produced independently from each other, and together they constitute a higher-level object: a software

product. Regardless of which one of these notions we commit to, the same making out of activities brought into existence the same software artifact. As I explained before, I think that Wang et al. argue for i). For this reason, although I think the rest of my examination applies to each of these cases, I will focus more on i).

In the case of a replacement at any given point during a software product production, under the condition that the replacement has taken place within the nested intentions which lead the overarching action of making out of the given software artifact, the given software artifact survives through these changes. For example, our programmer can produce two different programs, TickEd and TickADD, each of them the product of different making out of activities which are led by different intentions and performed by different individuals. They can share the same algorithm, the same code base, etc. Even if these two programs shared the exact same program specification, they would have been different and distinct programs. The reason again is that there would be two different making out of activities, and each of these making out of activities would necessarily bring into existence different programs. Our programmer can make a higher-level object out of one of these programs such as a software system out of the program TickEd. Then, she can replace TickEd with TickADD. In this scenario, our software system is made$_E$ out of TickEd, and before the replacement, the software system made$_S$ out of (constituted by) TickEd too. However, after the replacement our software system is made$_S$ out of (constituted by) TickADD. Since TickADD is related to TickEd by the ancestral of replacement, our software system changes its constituent and survive through this change (Evnine, 2009, p. 212). As a reminder, when there is such a replacement in Wang et al.'s view, at which point the former program constituted the given software

system was not clear. Moreover, how the software system was constituted by any of these programs was not clear either. Finally, these problems caused, from a metaphysical point of view, an odd conclusion, violation of the law of transitivity of identity.

Moreover, Evnine allows collective intentions and collective actions. In his account one software program can be produced by different programmers as a result of their collective activities. One programmer can suggest the algorithm of a digital artifact, another programmer can write the code base of the software product, a third one can define the external specification, etc. The end product would come into existence as a result of their collective actions. In the case of collective intentions and actions, one individual's workload might have been carried by another individual. For instance, the external specification of the program could have been defined by a fourth individual, instead of a third. The reason for this is that these people act in accordance with shared nested intentions to produce an artifact (Evnine, 2016, p. 236). (In Wang et al.'s view, software artifacts can also have multiple producers).

Therefore, in Evnine's account at the end of a complete programming process, there is one end object —a software product— which is made$_S$ out of an algorithm and/or out of some plural software artifacts if we were to accept different software artifact kinds which are argued by Wang et al. As I argued throughout this chapter, in Evnine's view, Wang et al.'s argument about layered constitution relations holds amongst software artifacts such as a program, a software system, a software application, etc, but under the condition that each of these artifacts are made out of one another.

Moreover, we can also reject Wang et al.'s claim about software artifacts are related to each other with layered constitution relations while accepting that a software product is constituted by several distinct software artifacts such as programs, software systems, software applications, etc. As a result of each of these cases, a software product is made out of these distinct software artifacts. In each of these cases, a software product can survive a replacement of its algorithm, or a replacement of one of its constituters, as long as its programmer(s) perform(s) the same overarching making out of activity.

For example, the algorithm of a given software product can change under the conditions that the producer(s) of the product carry the same intentions which they had when they are producing the product, and they are performing the same actions of making. The product can change its algorithm, its code base, its program, etc. but as long as these changes are part of the same making out of activities the product survives such revisions.

We can still reach both versions of the TickEd program: one with the Selection Sort algorithm, another with the Merge Sort algorithm. Evnine thinks that this is not a problem. His discussion of musical works and how they survive through change can shed light on this issue. Evnine allows certain changes in the sound structure of a musical work. For example, Lady Gaga can revise her work Poker Face, she can add some instruments, or change some melodies, or some rhythms to the song, etc. At the end she would reach a different sound structure, however, this new sound structure would have constituted the same work. A composer can make some changes while she is composing a work, and after she composes a work, and/or she publishes her work. The idea is that the composer still continues the same

composing process. When someone plays the so called earlier version of the work, he plays a part of the history of the work rather than the work itself. The composition exists as the current musical work and what called as an earlier version of the musical work exists in another form (Evnine, 2016, p. 138). Evnine thinks that changes concerning both the compositional process and the finished product still correspond to the same making out of process. Hence, the acts of revision of the composer are also part of the same overarching composing act under the condition that the composing intentions of the composer are consistent with her intentions to make certain revisions. Evnine thinks that the situation is similar to the practice of exhibiting finished paintings with their sketches and drafts (Evnine, 2009, p. 138). Similarly, our two versions of the TickEd programs can be considered as one and the same work. In this way, when someone choses to use the earlier version —the one which was constituted by the Selection sort algorithm— it would be the case of the usage of a part of the history of the work, instead of the work itself.

The challenging part is whether a software product can survive through changes in its high-level requirements or not. The high-level requirements are the descriptions of the features and functions of software products. The programmer makes the given software with an intention to satisfy these requirements through a machine. The initial intentions which leads the making out of activities of a programmer is represented by the high-level requirements of a software product (Wang et al., 2014, p. 328). Given that these requirements are descriptions of a programmer's creative intentions concerning the functions and features of her software product, any change in the requirements or any additional functions in the product prima facie seems to bring a new product into existence.

Evnine's discussion in the case of musical works and how they survive through change can illuminate this issue. As I just mentioned, Evnine allows certain changes in the sound structure of a musical work. A composer can make some changes after she composes and publishes her work.

Moreover, in Evnine's account abstract artifacts are modally flexible. Musical works can be produced at different times. The same musical work can be made out of another sound structure. Two composers can have make different musical works out of the same sound structure. The model properties of a musical work and the identity of a work are determined by its composer's composing acts and the compositional history of the work. The emphasis should be on these aspects in order to determine the modal possibilities of a musical work rather than on the work itself or a general notion of the ontology of musical works (Evnine, 2016, p. 139).

The identity of a work is mostly determined by the making out of activities of the producer of the given work. Another fundamental part is the content of the work as I mentioned above with regards to ready-made objects. The content of a musical work and its place in the repertoire of its composer has a role in determining the identity of the work, and the modal properties of the work. However, these roles are important because Evnine thinks they can affect the creative intentions and actions which bring the work into existence. There are some restrictions to the modal flexibility of the work. Evnine thinks that two different individuals cannot produce the same work because intentions and actions that bring certain works into existence necessarily belong to whom they actually belong to. As I mentioned before, the important point is that he does not argue that artifacts are rigidly dependent on their producers which means whenever there is a certain artifact it is necessarily produced

by whom it was actually produced. The result seems to be the same, but, he thinks that two different individuals cannot perform the exact same actions because two different individuals cannot have the exact same mental states and the exact same intentions (Evnine, 2016, pp. 245-246). If it were possible for two different persons to make the exact same actions, then these two persons could have produced the same artifacts. Evnine thinks that it is not possible for two different persons to have the same intentions and perform the same creative acts. For this reason, he thinks that his account provides a claim for original authorship.

At this point, someone can bring about Currie's version of the twin earth[6] scenario against the authorship claim of Evnine. According to Currie's example, there is a twin earth which is perceptually indistinguishable from our world, and the twin earth contains distinct individuals. Each person in our world has a twin in this earth. Under the condition that everything in this twin earth is the same with our actual world except individuals in these two worlds (Currie, 1989, p. 62). In our world we have Lady Gaga, and the song Poker Face which is still made out of its actual sound structure by Lady Gaga. Twin Lady Gaga* in this twin earth also composes twin Poker Face* in there. Assume that, everything is the same, the intentions of Lady Gaga, and twin Lady Gaga* concerning producing these musical works— Poker Face and Poker Face* are the same. The content and the historical producing process of these two songs Poker Face and twin Poker Face* are the same. Moreover, each of these musical works are made out of the same sound structure. Everything is duplicated but the composer of Poker Face in our world is Lady Gaga, and the composer of twin Poker Face* in the twin earth is twin Lady Gaga*.

[6] Twin earth arguments introduced by Putnam (Putnam, 1975).

Furthermore, in this twin earth, twin Lady Gaga* performed every compositional act exactly in the same way that Lady Gaga did, there is no difference between their actions. If we accept that Poker Face and twin Poker Face* are the same musical work, then the same musical work will be produced by two different individuals. If we accept the possibility of a twin earth where everything is the same, except individuals, then Evnine does not have a basis to argue for his authorship claim.

Evnine can argue against Currie's twin earth argument by insisting that if Lady Gaga and twin Lady Gaga* are two different individuals, then twin Lady Gaga* cannot have the exact same intention by which Lady Gaga has composed Poker Face. Therefore, Poker Face in our world is a different musical work from Poker Face* in the twin earth. However, if Evnine chooses to argue in this way, he needs an explanation as to why twin Lady Gaga* cannot have the same intentions with Lady Gaga. In Evnine's current argument, two different individuals cannot have the same intentions because intentions are mental states and they belong to whomever they actually belong to. In the case of twin earth scenarios an individual and her twin share almost every physical and psychological feature with each other, and they are numerically different. Hence, under these conditions, I do not see any reason for them not to have the same intentions.

The second way Evnine can argue against Currie is to say that Lady Gaga and twin Lady Gaga* are not different individuals, and for this reason Poker Face and twin Poker Face* are the same work. However, in this response Evnine would have to reject the metaphysical possibility of having a twin earth where everything is the same with our world, except the individuals in it (Currie, 1989, p. 67). I think Evnine should either accept that it is possible that the same musical work can be made out of

two different individuals, or he should reject the possibility of the twin earth scenario where this is possible. Given that twin earth scenarios require him to give up upon his claim on authorship of artifacts, I think he should reject the possibility of twin earth scenarios. However, I do not have an argument about how he can reject these scenarios. Moreover, Currie's argument is a problem for any account which has a claim on authorship about musical works.

Another restriction in his account is that an already existent musical work such as Lady Gaga's Telephone would not have the same structure with her other work Bad Romance. The reason is that the intentional producing activities of Lady Gaga which occurred when she composed Bad Romance is not the same with her intentional producing activities that occurred when she composed Telephone (Evnine, 2016, p. 139). Lady Gaga can make changes in her works such as changing some tones, altering the melody of the given work, or adding/ subtracting some notes. At the end of these activities, she ends up with a different sound structure from her initial work. However, as long as the creational acts involved in her compositional process of the given work are the same, the work which is produced through these acts is also the same (Evnine, 2016, p. 139). Moreover, if Lady Gaga were to compose Telephone much earlier in her carrier, it would have been a different work. The reason for this is that her creative activities that would have been involved in the compositional process of Telephone would have been different than her actual creative intentions which brought the song into existence (Evnine, 2009, p. 216). Therefore, the content and repertoire related restrictions are also concerned about the making out of activities.

Now, let us go back to the argument about software products, and whether a software product can survive through changes in its high-level requirements or not. In the case of Instagram, the product has been released in October 2010 exclusively on IOS. After two years from the initial version, a version for Android was released. In the same year with the Android version of the product, a feature limited website interface was released. The website version of the product does not allow its users to share and edit videos. However, both the IOS version and Android versions of the program allow these functions.

In Wang et al.'s view, each of these Instagram products are distinct and different objects. The website version of the product is different from the IOS and Android versions of Instagram because the requirements of the website version is different than both the IOS version and the Android version of the product. Moreover, even though the IOS version and the Android version of the product include all of the functions that the website version of the product has, the website program's functions are limited, and its requirements do not include all the same functions of the IOS and the Android versions of the product. The IOS version and the Android version of the product are different form each other because each of these software products were produced for different specific machines. The IOS version of the product was produced for some Apple products, and the Android version of the product was produced for Android based mobile phones and tablets. Therefore, each of these objects has different essential properties, and they are different from each other.

I think in Evnine's view all these versions of the Instagram are the same digital object. Given that each of these adaptations are seemingly consistent with the

intentional producing activities of producers of Instagram, they are produced during the same making out of activities, they are the same objects. The same programmer - or in the case of multiple producers, certain programmers- can produce each of these versions of Instagram throughout the same making out of activities with the intentions to produce and/or revise the same software product. Moreover, they can make some other changes with an intention to add some functions to their product. Furthermore, they can make their product available for different interfaces, and different machine systems too. The same product, Instagram, can survive through each of these changes because, each modification activity constitutes the same overarching making out of action. Since the identity of the work is determined by these making out of activities, each version of the Instagram product is the same object. The reason that we can have access to these different versions at the same time is that Instagram is an abstract object. We can reach different versions of the product from its producing history.

However, the high-level functions of Instagram, or the algorithm and the code of the product cannot be the same with another product like Facebook, or Twitter. In the case of these products, each was produced by different and independent producers. However, even if it were the case that they were all produced by the same producer, they would not be the same product. The reason for this is that each of these products are produced through different and distinct making out of activities. Each of these making out of activities necessarily brought into existence different products.

Moreover, the composing time and the historical context in which a musical work is composed are important in determining the aesthetic properties of the given

musical work. Similarly, technological developments, and advance in the machine systems on which digital objects will be run are important in determining some functional features of a digital artifact. For example, the producer of the Instagram product would not have produced the exact same product, when there were not any smart phones, and/or any other technology to inspire a programmer's intentions to produce Instagram. Furthermore, any digital artifact necessarily requires some machine systems, and/or some internet connection in order to work properly. I believe that without the existence of these kinds of technologies, we would not have the same digital artifacts we use today. However, I do not think that this pre-requisition implies that a digital artifact is rigidly constantly dependent on a specific kind of machine system which it is actually dependent on. Again, the same making out of activities which brought into existence of the product Instagram could have been occurred if Instagram would have been produced to work exclusively on a different machine system than it has been actually produced for. The reason for this is that both the intentions and the content of the making out of actions that bring Instagram into existence would be the same. The making out of activities by which Instagram was brought into existence could have had a different constituent —a different machine system requirement— in its high-level requirements.

I think Evnine's account of constitution is applicable to digital artifacts in the way in which I have explained throughout this chapter. As a consequence of this application I think we can have an account which can correspond to our linguistic and non-linguistic behaviors towards digital objects. I described three items to investigate the relations between two digital artifacts in different situations regarding their identity conditions. As a reminder they are listed below again:

1) When the user preface, the function, and the algorithm of two digital artifacts are the same but they have different and independent creators, and each of these artifacts are produced independently from each other.

2) When two digital artifacts have different algorithms and different texts but they carry the exact same function, and they are created by the same programmer as the same digital object.

3) When a digital object has changes in its user prefaces, or in its algorithm due to updates, and thereby gains or loses some functions or features.

According to Evnine's account we can claim that the objects described in item 1 are different from each other since both of these objects are produced through different making out of activities. However, Currie's twin earth argument is still an objection in this part. The objects described in item 2 are the same objects under the condition that these two digital artifacts are the products of the same making out of activities. The objects in item 3 are the same objects under the exact same condition with item 2.

4.4 An issue about revision

There is an issue about the revision of musical works and digital artifacts. As I argued above, according to Evnine, the making out of processes of artifacts includes the revision processes of artifacts too. Moreover, I also explained Evnine's argument that making out of activities should be performed by whomever they are actually performed by. Under these two conditions it prima facie seems that revisions of musical works and digital artifacts can be exclusively made by their actual producers.

On the other hand, Evnine also argues that an artifact can be produced by multiple individuals who share a nested intention in order to perform an overarching action together. The revisions made by those other than the original producer of a given artifact might also be considered as multiple individuals act upon nested intentions to perform an overarching action. In this case, the original intentions which actually brought into existence the given artifact can be understood similarly in the case of determining their modal properties: by investigating their history of production.

The situation might be considered similar to the Ship of Theseus puzzle. Evnine argues that in the Ship of Theseus case, the boards of the original ship are gradually removed, and new boards added to the ship. According to Evnine, during this removing process the ship remains the same while its matter has been replaced (Evnine, 2009, p216). Moreover, he thinks that in the case that someone has collected all the boards of which constituted the original Ship of Theseus before the replacement, and she makes a new ship out of these boards (Evnine, 2009, p. 216). The creative acts involved in the process of making this second ship is distinct from the making out of activities of the original Ship of Theseus. Hence, the second ship is not identical with the Ship of Theseus. Evnine thinks that there are two ships that are made out of the same matter at different times. These two ships must exist at different times because the same physical matter cannot be at two different places at the same time (Evnine, 2009, p. 216). In the case of abstract objects, there is no restriction (Evnine, 2009, p. 216).

In the case of physical objects such as the Ship of Theseus, the intentional producing activities, and the history of the work might be considered observable

through the work itself. However, I think in the case of abstract objects, the same observations are available through tokens -which are physical instances of an abstract object- of the given work.

Moreover, both physical objects like the Ship of Theseus and digital artifacts like software products might require some modifications through time. In the case of the Ship of Theseus, some restoration and modification might be required in order to keep the ship stable or to keep it functioning. For example, some planks and nails of the ship can wear off, or its painting can be peeled off. The boards of the ship can require replacements, etc. Since the Ship of Theseus is a physical object, one can easily observe that through these modifications, the ship continues to exist. However, if it were the case that after these modifications the end object is nothing akin to the original Ship of Theseus, then these modifications would have brought scarp into existence instead of the restored Ship of Theseus.

In the case of digital artifacts, although their matter cannot wear off like the matter of Ship of Theseus, a digital object can require some modifications, revisions, updates, and/or bug-fixing due to technological advancements. There can be new stakeholder requirements, machine system requirements, etc. In cases like these some producers might want to revise an already existent digital object in order to keep it in usage, or use it on different devices etc. For example, a programmer other than the actual programmer of Instagram might want to modify the Instagram product in order to make it available in a different machine system, or she might want to add a new function to the program. She can understand the original intentions which brought Instagram into existence by examining a copy of the Instagram. As long as her intentions concerning the revisions she applies to the Instagram program are

consistent with the original intentions which brought Instagram into existence; I think she can revise the product. In this case, I think her intentions and actions might be considered as a part of an overarching action which consists the intentions and actions of the actual producer of Instagram. Hence, our programmer's revisions might be considered under the making out of activities of Instagram. However, if our programmer's intentions are not consistent with the original intentions which brought Instagram into existence, then our programmer does not revise Instagram. If she intends to do so, she fails and produces scrap. Since her producing intentions have not been satisfied, she fails to produce a new artifact. Even though she intends to produce another software from Instagram, and successfully she creates a new object, it is not an original one. Moreover, I think other restrictions concerning the changes of musical works also apply in the case of these kinds of revisions, if we were to accept these revisions are possible to begin with.

I think both the ship of Theseus and a digital artifact can be revised by individuals other than their producers with an intention to keep them in usage, as the same artifacts which their producers produced. It might be thought similar to restoring a painting.

However, I think in the case of musical works, there is no requirement about adapting the musical work to some developments or to keep a musical work in usage by effecting some changes on the musical work. Moreover, if we were to accept that abstract objects such as digital works can be revised by some individuals other than their producers, I do not see a plausible reason to exclude musical works. Again, the same restrictions of the original intentions of the producer and the content dependent restrictions of modal flexibility of artifacts should be applied. For example, Pyotr

Tchaikovsky composed his final symphony in 1893. He intentionally included a sound illusion into his Symphony no6. The sound illusion concerns dividing the theme of the symphony into two different violins, and then locating these violins at different sides of the stage. Then he divided the accompaniment of the Symphony into two and distributed them into these same violins. Hence, both the theme and the accompaniment of the Symphony would be played piece by piece by these two violins, instead of two different melodies played by separate violins (Deutsch, 2019, p. 35). After Tchaikovsky conducted his symphony, a conductor Arthur Nikisch suggested to merge the theme together and play it with one violin, instead of two, and do the same merging for accomplishment and play it with another violin. Nikisch basically wanted to remove the sound illusion. However, Tchaikovsky refused to revise his work. A few days after their argument, Tchaikovsky died. Nikisch revised the work and initiated a new tradition to play it in his own revised version (Deutsch, 2019, p. 35).

I think according to Evnine's view what happens in the case of Symphony no6 is not a revision of the work because, Nikisch's intentions explicitly contradicts with the intentions which brought Symphony no6 into existence. Hence, they cannot be a part of the making out of activities of Symphony no6. I think, Nikisch's work is a distinct composition which used mostly the same sound structure with Tchaikovsky's Symphony no6. However, it is a different musical work from Symphony no6 because the intentions and making out of activities involved in the process of its production are different (and even inconsistent with) from the intentions involved in the composing process of Symphony no6. If Nikisch would

have adopted these changes in accord with Tchaikovsky's intentions, then his work would have been a revision of Symphony no6.

To sum up, I think Evnine's account can allow revisions of abstract artifacts, under the condition that the intentions which bring into certain revisions are consistent with the intentions which were involved in the producing process of the given artifact. Moreover, I think that the same restrictions concerning the modal properties of the work should be taken into account if we are to allow such revisions.

CHAPTER 5

CONCLUSION

In this thesis I attempted to defend a view which provides tools for identifying digital objects as either the same or different in accordance with our ordinary practices. I attempted to investigate the nature of digital artifacts. I examined three accounts by focusing on how these accounts argue for the existential dependence claim for abstract artifacts.

First, I examined Irmak's account. Irmak argues that the producers of abstract objects are causally responsible for the production of these objects. Moreover, abstract objects are historically dependent on their produces. Irmak does not provide an argument as to how the nature of digital artifacts are dependent on their producers. Moreover, he provides a strict identity criterion which does not allow change in abstract artifacts.

Second, I examined Wang et al.'s account. They argue that both the existence and the nature of digital artifacts are dependent on their producers, more specifically, certain creative activities of their producers. However, their account fails to explain how we can identify certain digital artifacts, and how different digital artifacts are related to each other. I argued that the constitution relation they have adopted does not apply to the objects in their account. Moreover, the application of their adopted constitution view causes serious ontological problems.

Third, I examined Evnine's account of constitution. He suggests a two-folded theory of constitution in which he argues for origin-as-act theory. In his account, both

the identity of artifacts and the nature of artifacts are dependent on the making out of activities of these objects.

I argued that Evnine's view can solve the problems of Wang et al.'s view because main problem with Wang et al.'s view is the constitution relation which they have adopted. They adopt Baker's view. Baker's view applies to material objects of different kinds. It relies on the genuine unity of an object and its constituent. The view requires spatial coincidence. I argued that Wang et al.'s view fails to satisfy each of these requirements.

Finally, I applied Evnine's constitution relation to digital artifacts. According to this application, digital artifacts are abstract objects which are brought into existence by their producers through certain intentional producing activities. Their identities and modal properties depend on the intentional making out of activities of their creators. I think, the application of Evnine's constitution notion provides a plausible account on the nature of digital artifacts in accordance with our linguistic and non-linguistic behaviors. In this way we can explain the changes in the digital artifacts.

There are two worries about Evnine's notions. First, Currie's twin earth scenario holds against his claim about the authorship of abstract objects. Second, Evnine relies on concepts like the discovery of an abstract entity such as sound structures, (and in the case of my application of his view the discovery of algorithms), and/or creation through mere thoughts and talk. However, he argues that such commitments are kind specific. Although they are applied to works of art, they are not applied to daily objects since daily objects require different kinds of labour to come into existence.

I think the application of Evnine's view to digital artifacts has more advantages than disadvantages. Moreover, I think this application provides a better explanation of the nature of digital artifacts than its alternatives.

REFERENCES

Baker, L., R. (2004). The ontology of artifacts. *Philosophical Explorations*, *(7)*2, 99-111.

Baker, L., R. (2007). *The metaphysics of everyday life.* Cambridge: Cambridge University Press.

Caplan, B., & Matheson, C. (2004). Can a musical work be created?. *British Journal of Aesthetics*, *(4)*2, 113-34.

Currie, G. (1989). *An ontology of art*. New York: Palgrave Macmillan.

Deutsch, D. (2019). *Musical illusions and phantom words: How music and speech unlock mysteries of the brain*. New York: Oxford University Press.

Evnine, S., J. (2009). Constitution and qua objects in the ontology of music. *The British Journal of Aesthetics*, *(49)*3, 203-217.

Evnine, S., J. (2016). *Making objects and events a hylomorphic theory of artifacts, actions, and organisms*. New York: Oxford University Press.

Irmak, N. (2012). Software is an abstract artifact. *Grazer Philosophische Studien*, *(86)*1, 56-72.

Levinson, J. (1980). What a musical work is?. *The Journal of Philosophy*, *(77)*1, 5-28.

Levinson, J. (2011). *Music, art, and metaphysics: Essays in philosophical aesthetics*. Oxford: Oxford University Press.

Manber, U. (1989). *Introduction to algorithms: A creative approach*. United States: Addison-Wesley Publishing Company.

Moor, J., H. (1978). Three myths of computer science. *The British Journal for the Philosophy of Science*, *(29)*3, 213-22.

Putnam, H. (1975). The meaning of "meaning". *Philosophical Papers*, (7) 13, 131-193

Renear, A., H., Dubin, D., Wickett, M., K. (2008). When digital objects change-exactly what changes?. *Proceedings of the Association for Information Science and Technology*, *(45)*1, 1-3.

Thomasson, L., A. (2009). Artifacts in metaphysics. In A. Meijers. (Ed.), *Philosophy of Technology and Engineering Sciences*. (pp. 191-212). Amsterdam: North Holland Publishing Company.

Wang X., Guarino, N., Guizzardi, G., & Mylopoulos, J. (2014). Software as a social artifact: A management and evolution perspective. Conceptual Modeling. ER 2014. Lecture Notes in Computer Science *(8824)*. Cham: Springer International Publishing.