PhD THESIS DECLARATION

I, the undersigned

FAMILY NAME	EBRAHIM
NAME	Mahdi
Student ID no.	1612555

Thesis title:

Three Essays on Architectural Complexity and Organization of Innovation in Knowledge-intensive Firms

PhD in	Business Administration and Management
Cycle	XXVII
Student's Tutor	CAMUFFO Arnaldo
Year of thesis defence	2017

DECLARE

under my responsibility:

1) that, according to Italian Republic Presidential Decree no. 445, 28th December 2000, mendacious declarations, falsifying records and the use of false records are punishable under the Italian penal code and related special laws. Should any of the above prove true, all benefits included in this declaration and those of the temporary "embargo" are automatically forfeited from the beginning;

2) that the University has the obligation, according to art. 6, par. 11, Ministerial Decree no. 224, 30th April 1999, to keep a copy of the thesis on deposit at the "Bilioteche Nazionali Centrali" (Italian National Libraries) in Rome and Florence, where consultation will be permitted, unless there is a temporary "embargo" protecting the rights of external bodies and the industrial/commercial exploitation of the thesis;

3) that the Bocconi Library will file the thesis in its "Archivio Istituzionale ad Accesso Aperto" (Institutional Registry) which permits online consultation of the complete text (except in cases of temporary "embargo");

4) that, in order to file the thesis at the Bocconi Library, the University requires that the thesis be submitted online by the student in unalterable format to Società NORMADEC (acting on behalf of the University), and that NORMADEC will indicate in each footnote the following information:

- PhD thesis: Three Essays on Architectural Complexity and Organization of Innovation in Knowledge-intensive Firms;
- by EBRAHIM Mahdi;
- defended at Università Commerciale "Luigi Bocconi" – Milano in the year 2017;
- the thesis is protected by the regulations governing copyright (Italian law no. 633, 22nd April 1941 and subsequent modifications). The exception is the right of Università Commerciale "Luigi Bocconi" to reproduce the same, quoting the source, for research and teaching purposes;

5) that the copy of the thesis submitted online to Normadec is identical to the copies handed in/sent to the members of the Thesis Board and to any other paper or digital copy deposited at the University offices, and, as a consequence, the University is absolved from any responsibility regarding errors, inaccuracy or omissions in the contents of the thesis;

6) that the contents and organization of the thesis is an original work carried out by the undersigned and does not in any way compromise the rights of third parties (Italian law, no. 633, 22nd April 1941 and subsequent integrations and modifications), including those regarding security of personal details; therefore, the University is in any case absolved from any responsibility whatsoever, civil, administrative or penal, and shall be exempt from any requests or claims from third parties;

7) that the thesis is not subject to "embargo", i.e. that it is not the result of work included in the regulations governing industrial property; it was not written as part of a project financed by public or private bodies with restrictions on the diffusion of the results; is not subject to patent or protection registrations.

Date ___31.01.2017___

ABSTRACT

This thesis focuses on the management of innovative activities in knowledge-intensive contexts. Specifically, it investigates how the product and organizational architectures co-evolve along the process of inter-generational technological innovations. It examines the drivers of such co-evolution, the underlying mechanisms, and its intertemporal dynamics. To accomplish such ambitious task, I meticulously collected micro-data of developing an industrial software over 60 months, during which 13 versions of the software were released.

The first essay of the thesis deals with the complexity growth of product designs across generations of innovation. There, I first examine the path-dependent evolution of complex designs, and then, investigate how such process is moderated by work organization. The findings suggest that in order to contain future design complexity, team work organization is more effective than individual task assignment, especially for *cyclical* design elements. This essay contributes to the innovation management literature, highlighting how path dependence and organizational choices jointly shape product architecture evolution, as well as how designers might prevent products and services from evolving overly complex.

The second essay uncovers the dynamics of the *mirroring hypothesis*—a well-established proposition maintaining that product architecture and organizational structure of the developing entities (individuals, teams, or organizations) mirror one another. It is well documented that misalignments between product and organizational architectures might undermine firm's competitive advantage. Thus, using a micro-dynamics perspective, I examine the misalignments between *technical dependencies* at the product level and corresponding *collaborations* at the organizational level, and investigate its antecedents. The findings suggest that misalignment increases when development tasks are routine, engineers are not co-located, and over time, as the deadline of releasing new software version approaches. These findings contribute to the organizational design literature, documenting how micro-organizational choices contribute to the misalignment between product and organizational architectures.

The third essay focuses on organizations' key decision of *search strategy* in their problem-solving activities. Specifically, it investigates how *problems' characteristics* influence the firms' decision of organizing search endeavors jointly or via independent individual efforts. My empirical findings demonstrate that the problem-solvers pursue joint search for solving complex problems (involving cross-domain interdependences), whereas they opt to independent search efforts when searching in distant loci (creating new features). Moreover, the results show that in the short term (during the development of each version), the engineers resort to joint search, whereas in the long term (along the subsequent versions), they turn to individual search expeditions.

Overall, the thesis contributes to innovation management literature by shedding light on the processes and dynamics of technological and organizational systems' co-evolution during continuous innovative activities and at the micro-level of study. It documents the path-dependent growth of a complex technological system and reveals the organizational contingencies of the process. It also explains how such organizational contingencies are endogenously shaped by innovative activities of problem solvers. Eventually, by focusing on the misalignment between organizational and technological systems, this thesis reveals how the co-evolution of the two interdependent systems unfolds as the synthetic outcome of the dialectical process of technological complexity growth and organizational uncertainty resolution over time.

ACKNOWLEDGEMENTS

Firstly, I would like to express my sincere gratitude to my advisor professor Arnaldo Camuffo for his continuous support, understanding, patience, and motivation all along my Ph.D. study and thesis writing.

Besides my advisor, I would like to thank Professor Carliss Baldwin for her generous support, encouragement and feedbacks during all these years. Furthermore, I would like to thank my thesis committee member Professor Carmelo Cennamo for his insightful comments, but also for the questions which helped me to widen my research to a broader audience.

My sincere thanks go to Dr. Daniel Sturtevant who provided me the technical guidance in data collection phase and shared with me his valuable thoughts and insights about the software development context. I am also indebted to Riccardo Fantato and Paride Ciatto from the company of my study, for their generous help and passionate taking time in my interviews, online calls, and emails.

In addition, I would like to thank Gautam Ahuja, Mary Benner, Gino Cattani, Gary Dushnitsky, Andrea Fosfuri, Lars Fredriksen, Marco Giarratana, Constance Helfat, Aseem Kaul, Andrew King, Tomi Laamanen, Tomasz Obloj, Gerardo Okhuysen, Claudio Panico, and Mellissa Schilling for their insightful comments and constructive critiques on different versions of the essays of this thesis. Errors and omissions, of course, are mine alone.

This thesis research was partially funded by The Cariplo Foundation, for which I am extremely grateful.

Last but not least, I am deeply thankful to my parents, whose support has been unconditional all these years. They have given up many things for me to be where I am; they have cherished with me every great moment and supported me whenever I needed it. Finally, and most importantly, I would like to thank my lovely wife Razieh. Without her support, encouragement, quiet patience, and unwavering love this thesis would never have been written.

Mahdi Ebrahim
31 January 2017

Table of Contents

ESSAY 1

Path Dependence and Organizational Choices in Product Architecture Evolution

ABSTRACT

Extant research highlights the significance of earlier design in shaping product architectural evolution and the tendency of product designs to become overly complex over time. However, the underlying mechanisms of such path dependence, as well as the impact of different organizational choices remain largely under-investigated. This study of an industrial software architectural evolution analyzes how past design characteristics affect the complexity of future product design, examining also how software development work organization choices influence such evolution. Our findings demonstrate that product architectures characterized by design modularity and hierarchy are less likely to evolve towards unnecessary complexity. Moreover, we show that, in order to contain future design complexity, team work organization is more effective than individual task assignment, especially for cyclical (non-hierarchical) design elements. We test our hypotheses on a unique dataset, which includes micro-level information about the architectural properties of 13 versions of an industrial software. Our study contributes to management of innovation literature, highlighting how path dependence and organizational choices jointly shape product architecture evolution, as well as how designers might prevent that products and services evolve overly complex.

Keywords: Modularity, hierarchy, cyclicality, architectural complexity, software development, team work, product design, path dependence

INTRODUCTION

The problem of designing complex products that over time are able to accommodate technological innovation central to technology management research (Ethiraj & Levinthal, 2004). Complex products are constantly updated and redesigned to adapt to ever changing human needs. Technological innovation—whether in the form of added product functionalities or enhancement of the existing product functions—brings about further complexity to product designs (Baldwin, MacCormack, & Rusnak, 2014; Baldwin, MacCormack, & Rusnak, 2012).

Adding or changing product features to incorporate new technologies might inadvertently interfere with other, unchanged parts of the product and impact them adversely, compromising overall product performance and requiring unforeseen adjustments in other components of the product system. Designing innovative, more complex products might therefore result in technical debt (Ramasubbu & Kemerer, 2014, 2015; Sturtevant & MacCormack, 2013) and cause a wide array of related issues including product malfunctions (Banker, Davis, & Slaughter, 1998), customers' dissatisfaction (Ethiraj, Ramasubbu, & Krishnan, 2012; Ramasubbu & Kemerer, 2015), designers' rework and stress (Sturtevant & MacCormack, 2013), and even increased employee turnover (Sturtevant & MacCormack, 2013; Sturtevant, MacCormack, Magee, & Baldwin, 2013).

Current product complexity deriving from past design decisions constrains designers' options and decisions about future developments. Once technical debt is accumulated because of past design decisions, this represents a burden on the development of next product generations. In this case, designers' work becomes particularly challenging, and often necessitates comprehensive and costly transformations of the product architecture, entailing the full redesign of components' boundaries and interfaces. Only if product designs remain evolvable, effective and efficient product development remains viable. Put differently, engineers have to design innovative products and make them more complex to meet customers' demand and incorporate new technologies, while at the same time prevent product designs from growing overly complex.

Scholars in the field of design and innovation suggest a set of instructions and rules to prevent product architectures from becoming unnecessarily complex over time (Baldwin & Clark, 2000; Sanchez & Mahoney, 1996; Simon, 1962). The two most important are design

modularity and design hierarchy, which have long been considered key principles to generate evolvable complex designs (Simon, 1962). In a modular design, maximum internal coherence and minimum external connectivity (Alexander, 1964; Parnas, 1972) enable flexibility, adaptability, and combinability in the design of the complex products (Baldwin & Clark, 2000; Cabigiosu & Camuffo, 2016; Ethiraj & Levinthal, 2004; Simon, 1962). Similarly, a hierarchical design composed of elements that are dependent on one another in a top-down hierarchy of dependencies, allows for parallel development activities on different elements at the same level, avoiding sequential interdependencies (Thompson, 1967) and reducing the need for coordination of development tasks. This, in turn, creates more flexible and evolvable products.

While "modularity" (or "near-decomposability") and " hierarchy" are the general principles to classify and evaluate the degree of design complexity (Alexander, 1964; Ethiraj & Levinthal, 2004; Parnas, 1972; Simon, 1962), recent research focuses on design microstructure (i.e. on design elements and their interdependencies) in order to provide more precise insights and more actionable knowledge about how to ensure that unnecessary complexity is restrained (MacCormack, Baldwin, & Rusnak, 2012; MacCormack, Baldwin, & Rusnak, 2007).

This study builds upon this stream of research on architectural properties and design micro-structures, and complements it in two ways: a) it offers a dynamic perspective, analyzing path dependence in product design evolution towards complexity; b) it integrates technological innovation and organizational design literature, investigating how micro-level organizational choices might affect the evolution of product architectures.

More specifically, after illustrating how previous architectural choices regarding the degree of modularity and hierarchy of design elements affect future architectural complexity, we focus

on how decisions about allocating tasks and deploying employees to problem-solving endeavors might affect the evolution of product architectures towards complexity and moderate the above described path dependence. Among these choices, one of the most important is whether to assign a given development task to an individual or a team of developers. This study therefore explores if and to what extent the architectural properties of a given product design affect its future evolution towards complexity, and how organizational choices—namely individual versus team task allocation—might moderate such path dependency.

We address our research questions analyzing a unique dataset of regarding the architectural evolution of an industrial software over the course of 60 months and across 13 different versions. Our dataset captures all the tasks performed on each file during the development of each version of the software.

Consistent with extant literature, we find that files characterized by modularity and hierarchy in previous software versions are less likely to become overly complex in the future. More importantly, we find that these effects are contingent on organizational choices concerning task allocation and performance. Team task assignment mitigates path dependence towards complexity, especially for non-hierarchical files. The effect is similar albeit smaller also in the case of non-modular files.

The paper is structured as follows. The second section lays out the theoretical framework, and states the hypotheses regarding the effects of design modularity and hierarchy, as well as work organization choices, on future product design complexity. The third section illustrates the data and methods focusing on modularity and hierarchy measures, the technical and organizational variables, the identification strategy and model specifications, and the dataset. The fourth section presents the findings, which are discussed in section five. The final section

highlights the study's contribution to the current debates in organizational design and management of innovation, clarifies the study's limitations, and offers some managerial implications and directions for future research.

THEORY AND HYPOTHESES

What explains the design evolution of complex products? Among different factors, the design choices of earlier versions of a given product are shown to play a key role, considering that the architectural properties of product designs tend to be inert (Baldwin et al., 2014; MacCormack et al., 2007; MacCormack, Rusnak, & Baldwin, 2006). For example, modular designs usually persist even after rounds of technological change (Baldwin et al., 2014). Similarly, product designs that are overly complex from the start are likely to remain complex or, more often, grow more complex over time, eventually necessitating complete redesign (MacCormack et al., 2007).

From this standpoint, the evolution of product designs can be considered a path-dependent process, i.e. a process which is non-ergodic, and thus unable to shake free of its history (David, 1985). Product design choices are limited by the decisions designers have made in the past and product development processes have path-dependent outcomes (Adner & Levinthal, 2001).

Generally, two distinctive design characteristics can parsimoniously characterize different architectural configurations (Ethiraj & Levinthal, 2004; Simon, 1962), modularity and hierarchy. A modular design includes nearly-decomposable clusters of design elements rendering maximum internal coherence and minimum external connections to other design elements (Alexander, 1964; Parnas, 1972). A hierarchical design involves elements that are interdependent on one another in the form of a top-down asymmetric hierarchy. Design

decisions pertaining to elements of upper levels impose restrictions on the functionality of lower level elements, hence constraining design decisions relevant to lower level elements (Baldwin & Clark, 2000, 2006).

Modularity and Design Evolution

A complex product comprises many design elements that are interdependent with one another to perform their own functions. Designing and managing products that tend to be more complex to satisfy more sophisticated market needs and incorporate new technologies is a challenging task for engineers with bounded rationality. The human's capability to memorize all the details of design elements and their interdependencies is limited and the information-processing requirements of managing complex designs can easily exceed engineers' cognitive capacity. The architecture of complexity literature suggests a set of design rules that, if implemented, allows to economize on bounded rationality helping engineers to make design choices that reduce the accumulation of unnecessary complexity. The first is to modularize the design by clustering the interdependent elements into self-contained modules, minimizing cross-module interdependencies and standardizing the corresponding cross-module interfaces. Modular designs are more evolvable because they allow "re-combinability," meaning that it can readily accommodate new modules (Cabigiosu & Camuffo, 2016). Modular designs are adaptive and flexible in that the comprising modules are nearly decomposed from one another, and therefore, any of the modules could accommodate internal changes without the need to correspondingly adjust other modules of the product. The propagation of changes is mostly contained within the boundary of the same module, and changes in the design element of a given module are less likely to affect elements belonging to other modules (MacCormack, Baldwin, & Rusnak, 2010). Modular designs are more adaptable to changes and less likely to result, through path-dependence in unnecessary product complexity as product design evolves

over time to incorporate technological changes (Baldwin & Clark, 2000; MacCormack et al., 2007; Sanchez & Mahoney, 1996).

Hypothesis 1 follows:

Hypothesis 1. Modular product designs are less likely to evolve into overly complex designs.

Our intuition, however, is that path dependence is not the only driver of future product design complexity. Rather, we believe that product architectures remain also the outcome of organizational choices, namely of how development work is organized. Path dependence from past designs (i.e. the extent to which they are modular) affects future design complexity, but such effect is moderated by how the development work is organized.

As previously mentioned, design modularity helps engineers to contain the propagation of changes inside a given module, and prevents them from affecting unintended parts of the product. Alternately, if a product design is less modular (more integral), a change in an element is more likely to unnecessarily impact the functioning of other elements. In these types of design, keeping the track of all the potentially affected elements might surpass the developers' cognitive capacity, which in turn could give rise to design issues (such as software bugs) and deflections in the overall product functioning, leading to further design complexity and incurring technical debt (Ramasubbu & Kemerer, 2015; Sturtevant & MacCormack, 2013; Sturtevant et al., 2013).

Engineers might not be able to recognize technical dependencies for two reasons—firstly, because they "hide" to their eyes (due to cognitive limits and design object complexity that, for example, makes design problem framing irksome (Brusoni & Prencipe, 2011)); and secondly,

technical dependencies might be created and/or changed because, in order to incorporate new technologies into the product design, dependency patterns between design elements must change over time. Put differently, bounded rationality and limits to cognitive skills and information processing make individuals unable to detect and consider large amounts and/or changing patterns of dependencies among design elements. So design modularity should be particularly helpful in preventing subsequent design evolution towards complexity when design work is organized individually (tasks are assigned to and performed by individual developers). Conversely, the benefits coming from teamwork (tasks are assigned to and performed by a team of developers) are negligible or less substantial in designing modular parts. Indeed, team-based organizations might even become inefficient because of the coordination costs associated with unnecessary information-sharing among team members about self-contained design elements. Such unnecessary communication might even paradoxically lead to more integral designs by generating unnecessary technical dependencies. Individual task assignments should therefore represent a better complement to design modularity.

Hypothesis 2 follows:

Hypothesis 2. The positive effect of design modularity on future product design complexity is stronger when development tasks are assigned to and performed by individuals rather than teams.

Hierarchy and Design Evolution

Rooting back to Simon's work on the architecture and evolution of complexity (1962), the role of *hierarchy* has been widely investigated in design literature. Since then, however, innovation and organizational design research have underplayed hierarchy while favoring modularity as desirable architectural property, so that the effects of design hierarchy on

complexity are comparatively understudied. Only recently, scholars have focused on design hierarchy's distinctive effects (Baldwin, MacCormack, & Rusnak, 2014), paying special attention to product designs' evolution towards complexity.

The impact of design hierarchy on the evolution of product architectures have been effectively analyzed by the stream of research about the *mirroring hypothesis* (Colfer & Baldwin, 2016; Sanchez & Mahoney, 1996). This proposition is grounded in organization theory and innovation literature (Baldwin & Clark, 2000; Henderson & Clark, 1990; Sanchez & Mahoney, 1996; von Hippel, 1990) as well as in software engineering research, where it is often referred to as *Conway's law* (Conway, 1968; Kwan, Cataldo, & Damian, 2012) or, more recently, *socio-technical congruence* (Betz et al., 2013; Cataldo, Herbsleb, & Carley, 2008; Herbsleb, 2007; Valetto et al., 2007). It suggests that the architectural properties of complex technological systems mimic the architectural properties of the social system that designed them. In the other words, social systems "mirror" technological systems, and the network structure of the former corresponds to the network structure of the latter (Brusoni & Prencipe, 2011, 2001, MacCormack et al., 2012, 2006; Sanchez & Mahoney, 1996). For example, imagine there is a symmetric reciprocal dependency between two design elements (e.g. two files in a software application). The *mirroring hypothesis* predicts that there is a reciprocal organizational tie between the two developers or development teams that have designed, developed, or changed each of the two elements (Colfer & Baldwin, 2016).

What drives the *mirroring hypothesis* is information-sharing requirements? Technical dependencies among the design elements of a complex product necessitate information-sharing among the developers of those elements. In order to adapt a given design element to the changes made in another technically-dependent element, it is crucial to share information about each of

the two elements between two separate developers, making sure that the two design elements are properly adjusted and continue to function seamlessly in tandem (Furlan, Cabigiosu, & Camuffo, 2014). The need for information-sharing is escalated when instead of having only two interdependent elements, there are many, simultaneously and reciprocally linked to each other, forming a *cyclic group* (therefore, we consider cyclicality as the inverse of hierarchy).

In a *cyclic group,* any change in one element requires exploration, experimentation, iterations of design, and rework, in order to readapt every element in the *cyclic group*, making each of them function well with the others (Baldwin et al., 2014). The fulfillment of such a challenging series of tasks generates a variety of issues and inefficiencies deriving from technical debt, including lower productivity and higher stress among the developers, and even higher rates of personnel turnover (Sturtevant & MacCormack, 2013; Sturtevant et al., 2013). Adjustment of highly interdependent design elements requires relentless communication among employees who work on design elements that belong to the same *cyclic group*. It is necessary to create a common understanding of the elements and their interdependencies, and it is critical to share it among the developers (Srikanth & Puranam, 2011). However, as the process of knowledge sharing among the developers is not immediate, takes time and resources, and requires learning (Srikanth & Puranam, 2014), failure to achieve a shared architectural understanding or incomplete sharing of the necessary information might lead to ineffective design choices, and consequently, to unnecessary complexity in architectural design, since developers might make changes to design elements (and the corresponding technical dependencies) without anticipating the effects.

Cyclic groups are sets of non-hierarchical elements (i.e. at least two elements linked by asymmetric and reciprocal dependencies that reverse the flow of dependency). These types of

dependencies juxtapose distinct pieces of knowledge embedded in separate elements of the *cyclic group*. To make these types of dependencies work seamlessly and contribute to the whole product's performance, the developers should carefully consider and design dependencies between all elements in the cyclic group, as well as re-consider and re-design *all of them* any time even only one is changed. This demanding task requires a lot of iteration and rework in designing, experimenting, and testing numerous variations in order to discover the optimal combination of design elements, and eventually gain the best performance of the product. Often such cognitively-challenging tasks could promptly get out of control when developers try to orchestrate all the interdependent elements to function properly in tandem, which demands learning about the functionality and dependencies of every single element in the *cyclic group*. As a result, these tasks lead to more haphazard design decisions, as the developers are overwhelmed by excessive information sharing, continuous task-coordination, and repeated design iterations, resulting in more complex designs (Ramasubbu & Kemerer, 2015). The situation gets even more complex when design elements belong to several cyclic groups simultaneously. Consequently, the evolution of product designs characterized by large cyclic groups of components (non-hierarchical designs) will generate path-dependent outcomes, i.e. unnecessarily more complex future designs.

Hypothesis 3 follows:

Hypothesis 3. Hierarchical product designs are less likely to evolve into overly complex designs.

However, engineers might not be able to recognize cyclic groups in product designs for two reasons: first, because they are not able to fully appreciate (again, due to cognitive limits and design object complexity) the nature of the technical dependencies between design

elements (e.g. mis-classifying them as sequential and not reciprocal (Thompson, 1967)); second, because they are not able to follow through the complete chains of direct and indirect dependencies that constitute a cyclic group, therefore making wrong design choices, especially if design elements need to incorporate new functionalities and new technologies in the product design. This might happen for a variety of reasons including time pressure, resource constraints, lack of capabilities of ill-designed development processes. Put differently, bounded rationality, as well as limits in cognitive skills and in information processing might make individuals unable to see the patterns of direct and indirect dependencies among design elements in cyclic groups. Consequently, focusing on design hierarchy is particularly helpful in preventing subsequent design evolution towards complexity when development work is organized in teams (tasks are assigned to and performed by teams of developers). Team-based organizations can be especially efficient if cyclic groups are contained in number and size, so that developers can focus on sharing only necessary information. They can also concentrate on problem-solving about reciprocal, direct, and indirect interdependencies that are actually relevant to product performance (those within a given cyclic group). Teamwork is therefore a better complement than individual task assignment to design hierarchy.

Hypothesis 4 follows:

Hypothesis 4. The positive effect of design hierarchy on future product design complexity is stronger when development tasks are assigned to and performed by teams rather than individuals.

DATA AND METHODS

A complex product comprises a large number of elements, each of which accomplishes a specific function. The proper functioning of the product as a whole is the result of meticulous

orchestration of elements that depend on one another to function seamlessly. Hence, architectural design of a complex product could be represented as a network where nodes represent design elements with specific functionalities, and ties represent interdependencies between them. Such a network can be usefully represented as a matrix in order to facilitate the definition and operationalization of design characteristics (modularity and hierarchy) at the design-element level. The matrix representation of a product design is generally referred to as a "design structure matrix" (DSM). In theory, a design element is fully modular when it depends only on elements inside its own module and therefore, to function properly, it does not depend on any element in other modules. Alternatively, a fully hierarchical design element does not hold any reciprocal (or cyclic) interdependence with other elements. It might unilaterally (sequentially) dependent on other elements, and other elements might depend on it unilaterally. A non-hierarchical element, instead, possesses cyclic dependencies with other design elements.

Design structure matrix (DSM)

Most of the methodologies used to represent the design of a complex system are based on the assumption that design can be well-modeled by recording the patterns of (directional) dependencies among its constituent elements (Browning, 2001; Eppinger & Browning, 2012; Smith & Eppinger, 1997). Design Structure Matrices (DSM) provide such representation of dependencies. The DSM is a square matrix where each element of the system takes one column and one row. Then, if element *A* depends on element *B*, we insert a *1* in cell *(A,B)* of the matrix. In other words, we put *1* on the *A*'s row and *B*'s column. The whole matrix then represents directional dependencies among all elements of the system. The figure bellow is an example of a design arrangement where, for instance, *A* depends on *B* and the dependency is represented in the corresponding DSM matrix.

Alternatively, element *A* is dependent on element *B*, and *B* in turn, is dependent on *C*. Then, *A* is indirectly dependent on *C*. To capture both direct and indirect dependencies in our analyses we use an extension of DSM, called *Transitive Design Structure Matrix* (TDSM), that represents both indirect and direct dependencies.

Insert figure 1 about here

Using DSM to represent design modularity and hierarchy

As above described, design modularity and hierarchy can summarize the degree of architectural design complexity (Ethiraj & Levinthal, 2004, 2002; Murmann & Frenken, 2006). A design is *modular* when it is nearly-decomposable into subsets of elements (Simon, 1962), rendering maximum internal coherence and minimum external coupling to elements of other subsets of the system (Alexander, 1964). A DSM representation of a modular design consists of square sub-blocks along the matrix's diagonal. The sub-blocks represent modules of the system and capture dependencies between elements inside the same module. If elements in different modules are not linked to each other, its corresponding visual representation in the TDSM would be blank cells outside the sub-blocks. As an example, the design configuration and its corresponding TDSM are presented in Figures 2 and 3.

Instead, a design is *hierarchical* when it does not involve any reverse (or reciprocal feedback) link between design elements. In such architectural configuration, a change in one element propagates to elements only downstream in the dependency chain, and does not affect any upstream design element. The graphical demonstration and corresponding TDSM of a hierarchical design is presented in Figures 2 and 3.

Insert figures 2 and 3 about here

Product designs are characterized by different degrees of modularity and hierarchy. These two architectural properties can be used to categorize different product designs conceptually locating them in the 2x2 framework presented in Figure 2. This framework facilitates thinking about the microstructure of any product design, as each design element is characterized by a different degree of modularity and hierarchy.

Data description

In order to empirically investigate how *design modularity* and *design hierarchy* contribute to mitigate unnecessary complexity growth of product design over time, we collected a unique design element-level dataset of an industrial software package, which evolves over a period of 60 months and along 13 versions. The software is developed by the software division of a leading and innovative HVAC (heating, ventilation, and air conditioning) company based in Italy with software developers located in Italy, China, and India. The main function of the software is supervising HVAC micro-controllers in plants. such as industrial fridges and supermarkets.

Our dataset comprises technical data of the architectural design of each software version, including the data of the system's elements (i.e. files) and their dependencies, in addition to information of changes made to each element during the development of each version of the software, and the date and time of each change. This rich dataset provides an opportunity to study the architectural evolution of a product design, as well as the dynamics and determinants of its complexity.

We extracted the technical data of software files and their dependencies by using a software package called Understand, which parses lines of code in all files, and extracts various types of dependencies between them, such as function calls, uses, reads, etc. In addition, the dataset contains information of every single change to the architecture of the software over the period of the study.

Moreover, the dataset records the type of development task performed on any given file at a given point in time, the developer who accomplished the task, and whether the task was assigned and performed individually or by a team of developers. This allowed to observe how a given design element changed across software versions, how its architectural properties changed as a result of certain types of development tasks, and how it eventually evolved, in terms of degree of complexity, contributed to the overall complexity of subsequent software versions.

Measures

Dependent variable

Design complexity is the dependent variable and we measure it—at the *design element-level*—as the number of dependencies (direct or indirect) a focal element (file) possesses at a given point in time (MacCormack et al., 2007). This measure is widely used to measure the complexity of various complex systems (e.g., Rivkin & Siggelkow, 2003; Zhou, 2013).

Independent variables

Design modularity. It is measured—at the *design element-level*—as the absolute number of cross-module dependencies, a focal element (file) possesses at a given point in time (reversely coded). Cross-module dependencies are identified as the dependencies between product modules, as derived from the application of the Louvain algorithm. We used the

Louvain algorithm (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008) to identify modules. This algorithm is based on modularity optimization and is shown to outperform all other known module-detection methods in terms of computational accuracy and time (Blondel et al., 2008).

The Louvain method consists of two phases. First it looks for small modules of design elements optimizing modularity locally. Then it aggregates nodes of the same module and builds a new network composed of these modules. These steps are repeated iteratively until a maximum degree of modularity is attained. The degree of modularity of a module is a scalar value between -1 and 1 that measures the density of dependencies within modules as compared to dependencies between modules. The degree of modularity Q, therefore, is defined as:

$$Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{K_i K_j}{2m}\right]\delta(c_i, c_j),$$

where A_{ij} represents the weight of the link between *i* and *j*, $k_i = \sum_j A_{ij}$ is the sum of the weights of the edges attached to vertex *i*, c_i is the module to which vertex *i* is assigned, the δ-function δ(u, v) is 1 if u = v and 0 otherwise, and $m = \frac{1}{2}\sum_{ij} A_{ij}$ (Blondel et al., 2008).

Design hierarchy. As already illustrated, design cyclicality—the extent to which design elements are linked by reciprocal dependencies—is the inverse of design hierarchy. Therefore, we measure design hierarchy using an inverse proxy: the size of the largest cyclic group to which the file belongs (if any). The larger the size of the cyclic group, the larger the number of design elements reciprocally dependent on the focal design element, the larger the degree of cyclicality, and the lower the degree of design hierarchy of the focal design element. We calculate the size of cyclic groups applying the algorithm suggested by Baldwin, MacCormack, and Rusnak (2014). This algorithm is based on a specific type of reordering of the Design Structure Matrix (DSM) of the whole software architecture, which allows to identify the non-

hierarchical dependencies. Sorting the DSM first descending, based on inward dependencies, then ascending, based on outward dependencies allows to identify the non-hierarchical dependencies (for a detailed review of the methodology and proofs, please refer to the original paper, Baldwin et al., 2014). The descriptive statistics of the dependent and independent variables are reported in table 2 in the results section.

Control variables

Following common practice in the software engineering literature, we control for heterogeneity in the content of the files. Controlling for technical, file-related variation of the dependent variable is necessary to isolate the effect of the proposed explanatory variables from other possible sources of variation, namely from heterogeneity in the development tasks deriving from file-specific characteristics.

We use a dummy variable, *Java language,* to control for the language in which the file is written (Java files=1, C and C header files=0). A second control variable, Lines of Code, *LOC,* is a software metric used to measure the size of a computer program by counting the number of lines contained in the files' source code. Lastly, *cyclomatic complexity* measures the internal (and not structural) complexity of the file and hygiene of codes written and saved in the file. We expect that these three file-related control variables increase the likelihood that files in future software versions will become more complex.

Model Specifications

To test hypotheses 1 and 3, i.e. the main effects of past design modularity and hierarchy on future file complexity, we use panel data regressions. The basic specification of our econometric model is the following:

$$design\ complexity_{it} = \beta_0 + \beta_1\ modularity_{it-1} + \beta_2\ hierarchy_{it-1} + X_{it-1}\mathrm{B} + \mu_i + \varepsilon_{it}$$

where: a) the dependent variable is design complexity measured as the total number of direct and indirect dependencies a file *i* has at time (version) *t*; b) the independent variables are design modularity, measured as the number of cross-module dependencies a file *i* has at time (version) *t-1*, and design hierarchy, measured as the size of the largest cyclic group a file *i* has at time (version) *t-1* (reversely coded). A set of control variables, recording technical characteristics of the files at time (version) *t-1*, is added to account for the proportion of variation in the dependent variable that is generated by other co-variates.

We first estimate the panel data regressions with files' random effects. Then, we add files' fixed effects to relax the randomness assumption and check whether the results are comparable. Table 3 reports the results of the analyses. Random and fixed effects estimations do not differ in size and significance.

Hypotheses 2 and 4 are instead tested using pooled regressions. These two hypotheses concern the moderation effects of individual versus team assignment of development tasks. In order to test these hypotheses, we move to a different unit of analysis, *file×task* in each version of the analyzed software. The reason is that multiple tasks might be performed on the same file during the development of a given software version. Nonetheless, in order to control for potential sources of unobservable variation in the dependent variable, we estimate software versions' and developers' fixed effects. Thus, we estimate the following regression equation:

$$\begin{aligned} &design\ complexity_{it} \\ &\quad = \beta_0 + \beta_1\ modularity_{it-1} + \beta_2\ hierarchy_{it-1} + \gamma_1 modularity_{it-1} \times task\ assignment_{it'} \\ &\quad + \gamma_2 hierarchy_{it-1} \times task\ assignment_{it'} + X_{t-1}\mathrm{B} + \tau_t + \lambda_d + \varepsilon_{it} \end{aligned}$$

wherein: a) the dependent variable is design complexity measured as the total number of direct and indirect dependencies a file *i* has at time (version) *t*; b) the independent variables are design modularity, measured as the number of cross-module dependencies a file *i* has at time (version) *t-1*, and design hierarchy, measured as the size of the largest cyclic group a file *i* has at time (version) *t-1* (reversely coded); c) the moderation variable is of the different type of work organization entailed by the nature of task assignments, which equals 1 if the given task on file *i* is assigned to and performed by a team and is equal to 0 if the given task on file *i* is assigned to and performed by individual developers, at time *t'* (during the development of version *t*, where $t\text{-}1 < t' < t$). Also in this case we control for the technical characteristics of each file *i* at time (version) *t-1*, and we use different model specifications which include software versions' fixed effects τ_t and developers' fixed effect λ_d.

FINDINGS

This section reports the results of the analyses and is articulated in four subsections. The first presents the descriptive statistics and correlations. The second presents the regression results regarding hypotheses 1 and 3, where the unit of analysis is the focal *file* per version. The third presents the estimates for hypotheses 2 and 4, testing the organizational contingencies of design evolution where the unit of analysis is *File×Tasks*. The fourth subsection reports a series of robustness checks that corroborate our findings.

Descriptive statistics

The dataset used in this study comprises 13 versions of an industrial software over the course of 60 months of development, during which the software continuously grew in size and complexity. Table 1 and graphs below document the complexity growth of the software across

subsequent versions. As it is evident, the number of design elements and the interdependencies between them systematically increased over time as it incorporated new features to accommodate for new technology and meet customers' demand. As expected, the data show that, once the company decided to design a completely new version of the software, albeit keeping updating and developing the existing one, the new version of the software represented a major discontinuity vis-à-vis the existing one. The level of complexity of the new stream of software (versions 2.X) is significantly larger than the level of complexity of the older versions (1.X), which corresponds to a completely new software architecture, set of functionality, number and type of features. Figure 4 reports some summary statistics about the software versions, both 1.X and 2.X. The pattern of the overall number of files, total direct dependencies and total indirect dependencies show the increase in the complexity of the software over time, with versions 2.X getting more complex, over time, compared to versions 1.X. Table 1 includes a wider set of descriptive statistics about the analyzed software versions.

Insert table 1 about here.

Insert figure 4 about here.

Table 2 reports the descriptive statistics for the main variables of the study. Overall, the dataset used for testing hypotheses 1 and 3 includes 29705 observations (files×versions). As already mentioned, to test hypotheses 2 and 4 we used a complementary dataset including additional data about the assignment and performance of development tasks on the files. The number of observations in this dataset is 11855 files×tasks×versions.

The dependent variable, design complexity demonstrates a reasonable range and heterogeneity. Considering the longitudinal nature of our research question, we used OLS panel data regressions to test the hypotheses 1 and 3. Further, also the independent variables demonstrate enough variation, and comparable ranges that facilitates the interpretation and comparison of their effects.

Insert table 2 about here.

Table 3 reports the correlation coefficients between the variables used in the regression models. The correlations between the variables that are used as co-variates in the regressions are low (below 0.3). This confirms that design modularity and hierarchy, although not perfectly orthogonal, capture different architectural properties.

Insert table 3 about here.

Regression results for path-dependent design evolution

Table 4 reports the results of four regression models estimating the effects of hypotheses 1 and 3. Model 1 and Model 2 estimate the effects of modularity (reversely coded using, as inverse proxy, the number of cross modular links) and hierarchy (reversely coded using, as inverse proxy, cyclicality, i.e. the size of the largest cyclic group) of design on future file complexity (i.e. the number of its dependencies in the next version of the software). The results of estimations with random effects provide support for our hypotheses. Both *design modularity*

and *design hierarchy* of a given file in a given software version decrease the file's complexity in the next version. The fact that the interaction effect of the two independent variables is negligible provides additional evidence that design modularity and design hierarchy are distinct architectural properties and influence design evolution through two distinct mechanisms. Further, the results indicate that the detrimental effects of the independent variables diminish for increasing levels of the two variables.

Similarly, Models 3 and 4 estimate the main effects of design *modularity* and *hierarchy* in a panel data with files' fixed effects. The results support the effect of modularity but the effect of hierarchy is no longer significant, though the estimated effect is positive, as hypothesized. The reason why the estimated coefficients are not as significant as in the random-effect model might stem from the fact that the observations at the file level are highly auto-regressive: the files, their interdependencies, and more generally the software designs are inert in most of the cases. In the development of new versions, usually minor chunks are added to the software while major parts are slightly changed and modified. Our findings in the next section supports this intuition as the results demonstrate significant effects for both design variables.

Insert table 4 about here.

Regression results for the moderation effects of work organization

To test the hypotheses regarding the moderation effects of *task assignment* on the design evolution we used pooled OLS regressions. Model 1 in table 5 reports the estimations for the main effects (H1 and H3), in addition to those of the control variables. Consistent with what

illustrated in the previous subsection, we found support for the main effects of design modularity and design hierarchy on the files' complexity of the next version.

Further, consistently with mainstream software engineering literature, we found that Java files, compared to C files, affect the complexity of software architectures (Baldwin et al., 2014). Finally, as expected, the internal complexity of the files and the code lines contained in them significantly increase the complexity of each file in the next version.

Models 2 estimates the moderation effects of different types of task assignment on the relationship between current file modularity and future file complexity. Model 3 includes the software versions' and developers' fixed effects. Model 2 estimates do not support hypothesis 2, while model 3 provides evidence in the opposite direction of that hypothesized. We will revisit this unexpected finding later.

Models 4 and 5 investigates the moderation effect of different types of task assignment on the relationship between current file modularity and future file complexity. We found significant support for hypothesis 4, suggesting that assigning development tasks on non-hierarchical files to teams of engineers (instead of individuals) helps reducing future file complexity.

Lastly, models 6 and 7 are fully fledged models including both moderation effects. Again, hypotheses 1 and 3 are supported. Moreover, hypothesis 4 is also supported.

Unexpectedly, the moderation effect regarding hypothesis 2 is not significant and in the opposite direction of what hypothesized. The mitigating effect of design modularity on future file complexity does not appear to be stronger in the case of *individual* task assignment. Nonetheless, comparing the two moderation effects one could conclude that the reducing effect

of assigning tasks to teams is remarkably more sizeable when the teams work on non-hierarchical files compared to when they work on non-modular files. Figures 5 and 6 provide a visual representation of the interaction effects and offers the size effects of the estimated coefficients. Figure 5 demonstrates the moderation effect of task assignment on the non-modularity/complexity relationship. While the estimated coefficient is significant, its size effect is overshadowed by the substantial direct effect of the modularity. On the contrary, the moderation effect of task assignment on the non-hierarchy/complexity relationship is meaningful and significant. Especially for highly-cyclical files, team task assignments considerably reduce file complexity in the next version, whereas individual task assignment on files with the same level of cyclicality detrimentally increases file's future complexity.

Insert table 5 about here.
Inset figures 5 & 6 about here.

Robustness checks

A series of robustness checks were conducted in order to examine the validity of results over partially-different identification strategies. As discussed earlier, we checked the interacting effects of the two main independent variables in the panel data regression. The fact that *design modularity* and *design hierarchy* have no interaction effect on the dependent variable confirms one of the key propositions in architectural complexity literature, i.e. design modularity and design hierarchy are two distinct architectural properties. Moreover, we tested the robustness of the hypotheses with software versions' and developers' fixed effects. Accounting for the versions fixed effects is necessary since the various versions are developed

for different purposes of incremental improvements or more radical changes and new feature developments. Furthermore, from our field visits we learned that team members with varying job roles behave differently while performing tasks, and hence, we are obliged to control for individual developers' fixed effects. The robustness checks conducted corroborate our previous findings.

DISCUSSION AND CONCLUSION

This study explains why product designs tend to become overly complex over time. Path dependence, in the form of the degree of modularity and hierarchy of past product designs, represents a major driver of future design complexity and of the associated negative effects in terms of technical debt, low development productivity, and dysfunctional organizational behaviors. Consistently with architectural complexity literature, we show that the degree of modularity of previous product designs has a larger potential impact than design hierarchy on reducing the evolution of future designs towards unnecessary complexity. However, adding to modularity literature, we show that organizational choices regarding development tasks might affect these path-dependent outcomes. Organizing software development works in teams, instead of assigning development tasks to individuals, moderate the above-described path dependence and curbs the tendency towards unnecessary complexity. Team work amplifies the effect of design modularity and design hierarchy in containing future complexity. This positive moderation effect is particularly large for design hierarchy.

If a team of engineers plan to perform tasks on interdependent design elements, then design hierarchy plays a key role in the task coordination and accomplishments. If the elements are connected to each other in a hierarchical design configuration, meaning that they are asymmetrically and unilaterally dependent on one another, then the optimal sequence of

changes is planned by examining the dependency chain and accordingly devising a set of actions. However, if the elements involved in the given task belong to a non-hierarchical design element characterized by cyclic groups and reciprocal dependencies, then, conducting development tasks requires many iterations and rework by engineers to simultaneously adjust reciprocally-dependent elements to new changes. This is a costly and time-consuming endeavor which, under time pressure might lead to inappropriate design changes which, in turn, might cause unnecessary complexity and technical debt.

We also found empirical evidence that while performing tasks on non-modular design elements, team task assignment is only marginally better than individual task assignment in reducing future file complexity. If an individual is assigned to the task of making changes on a given file, she should accurately predict the propagation of the change to other elements. If the focal file is modular (i.e. it has dependencies to files only inside its own module), then the individual developer can comfortably track which elements will be affected and effectively adjust the elements impacted by the new change. However, when the file is non-modular (i.e. there exists cross-module interdependencies), predicting propagation of changes becomes more challenging for the bounded rational individual. As a result, the developer fails to anticipate and adjust affected elements, which eventually give rise to unintended increased complexity in future versions.

Our findings on the micro-dynamics of complexity in product designs contribute to the technology and innovation literature by complementing existing macro-level literature on complexity evolution. Also, the paper contributes to the innovation management research clarifying the determinants of path dependence in product design evolution and identifying how organizational choice might curb such path dependence.

We show that complex designs might take on different evolutionary patterns, and while initial imprinting is extremely important (product architectures characterized by design elements that are modular and hierarchical since the beginning are unlikely to evolve into overly complex designs), the way in which development tasks are assigned and organized might make a significant difference in containing evolution towards unnecessary complexity.

The managerial implications of this study are straightforward. First, engineers should spend significant time monitoring the degree of modularity and hierarchy of product components early on, when they design, prototype and launch the first version of any product. Then, project leaders or chief engineers should carefully decide how to allocate responsibilities of development tasks, making wider use of teams especially in case of non-hierarchical components.

Design elements should be classified according to their level of design modularity and hierarchy. Elements that are both modular and hierarchical are easy to manage and the corresponding development activities are easy to organize, as they contribute least to future complexity. At the opposite, design elements that are neither modular nor hierarchical need to be completely redesigned to become more modular or more hierarchical. Our findings suggest that, to minimize future complexity, the best strategy is to assign development tasks regrading cyclical (non-hierarchical) elements to teams, while assigning development tasks on non-modular elements to individuals. This should result in evolvable designs.

This study opens new avenues for future research in technology and innovation literature. First, by building upon our conceptualization and operationalization of technological design at the element-level, future studies could study “imprinting” mechanisms and what prevents engineers to come up with product designs whose components are as modular and as

hierarchical as possible from the beginning. Second, they could uncover other organizational contingencies that might affect the evolution of product designs. Developers' and development teams' routines, design tools, collaborating technologies and incentives are good candidates for future studies. Moreover, in order to replicate and validate our findings in contexts other than the software industry, future research has to look into other innovation-intensive industries and reveal new contingencies, delivering novel insights.

Several limitations apply to this study. First, we examined organizational and technological changes in the context of a specific company, as well as the software industry. While the software development context is replete with frequent problem-solving on a daily basis, our findings might not be readily extended to other contexts with different routines and frequencies of problem-solving activities. Second, we studied a commercial software, developed in a closed-enterprise setting. Our suggested organizational contingencies, then, might not be fully applied in the context of open-source software development, or voluntary problem-solving and open organizations. Finally, most of the innovative activities studied in this research were incremental rather than radical. Our suggested implications have to be reexamined during instances of radical and disruptive (and most likely exogenous) technological changes, as a company's knowledge base is rendered obsolete, and therefore organizational contingencies of conducting innovation should be different.

FIGURES AND TABLES

Figure 1. An example of design configuration (top), represented by DSM (bottom left) and TDSM (bottom right), *(MacCormack et al., 2006)*

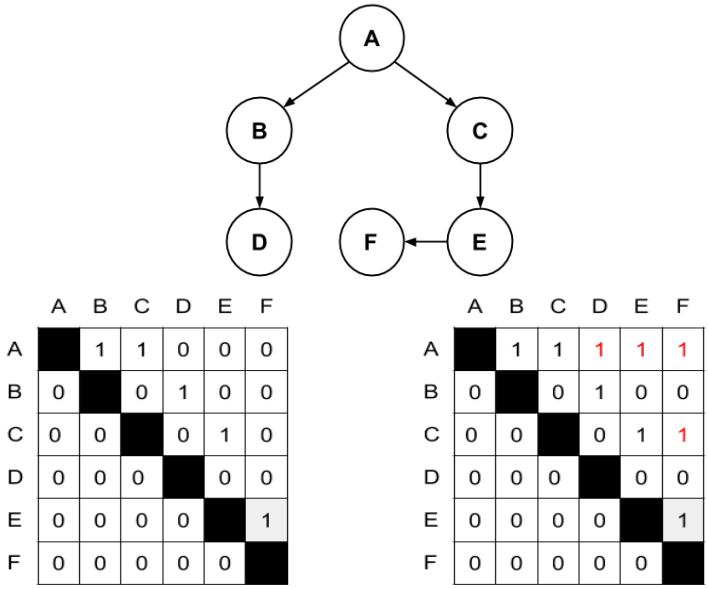


Figure 2. Design configurations categorized by modularity and hierarchy

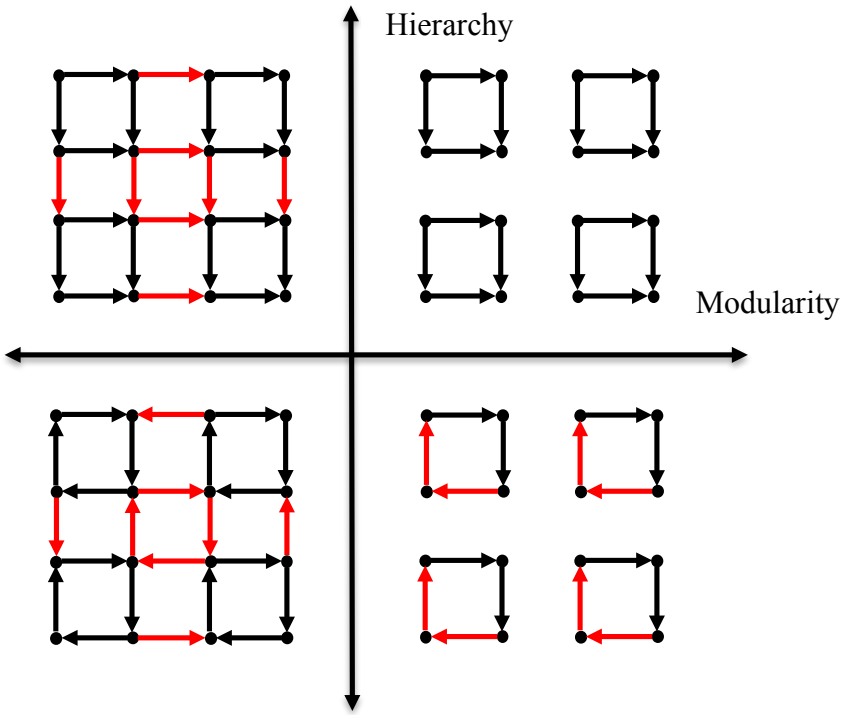


Figure 3. TDSM representations of design configurations categorized by modularity and hierarchy

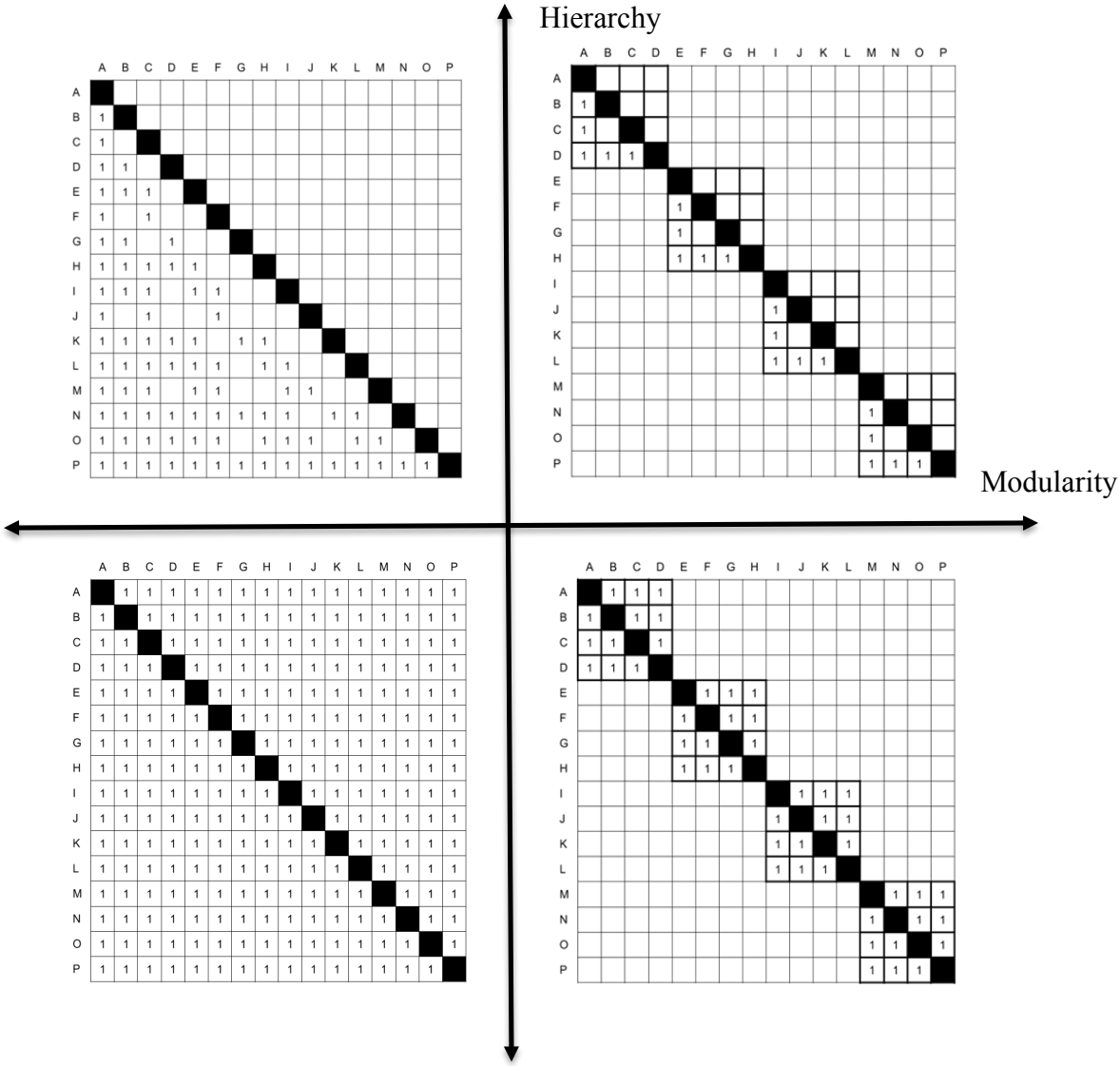
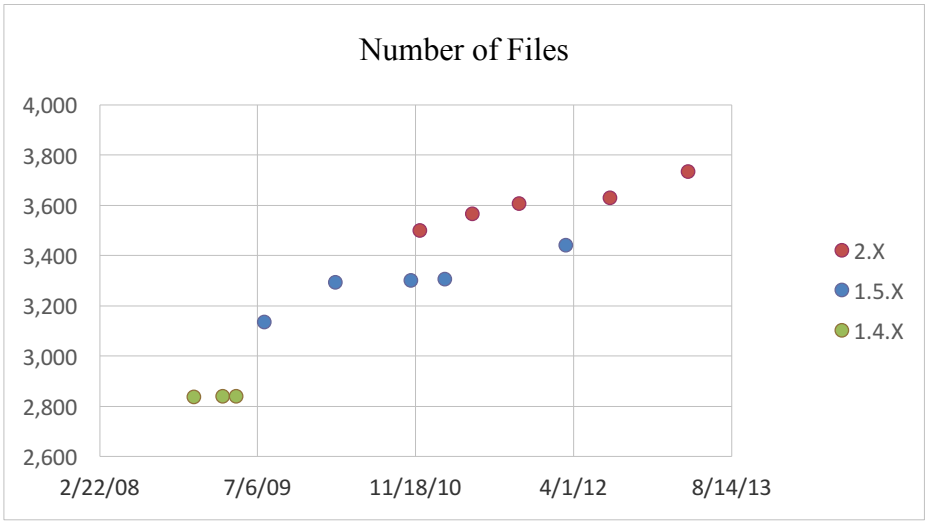
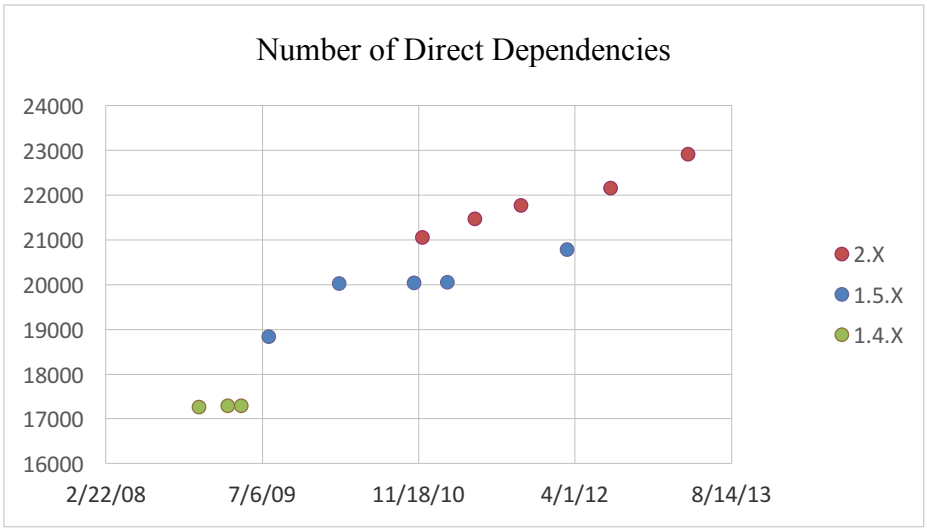


Figure 4. Complexity growth of the software versions over time





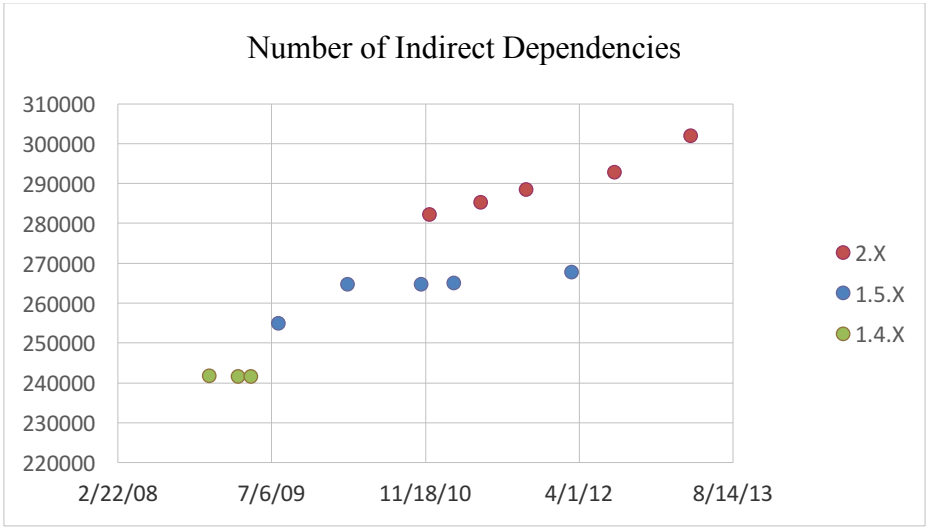


Figure 5. Differential effect of non-modularity on design complexity (at the end of the development phase) as contingent upon task assignment decision

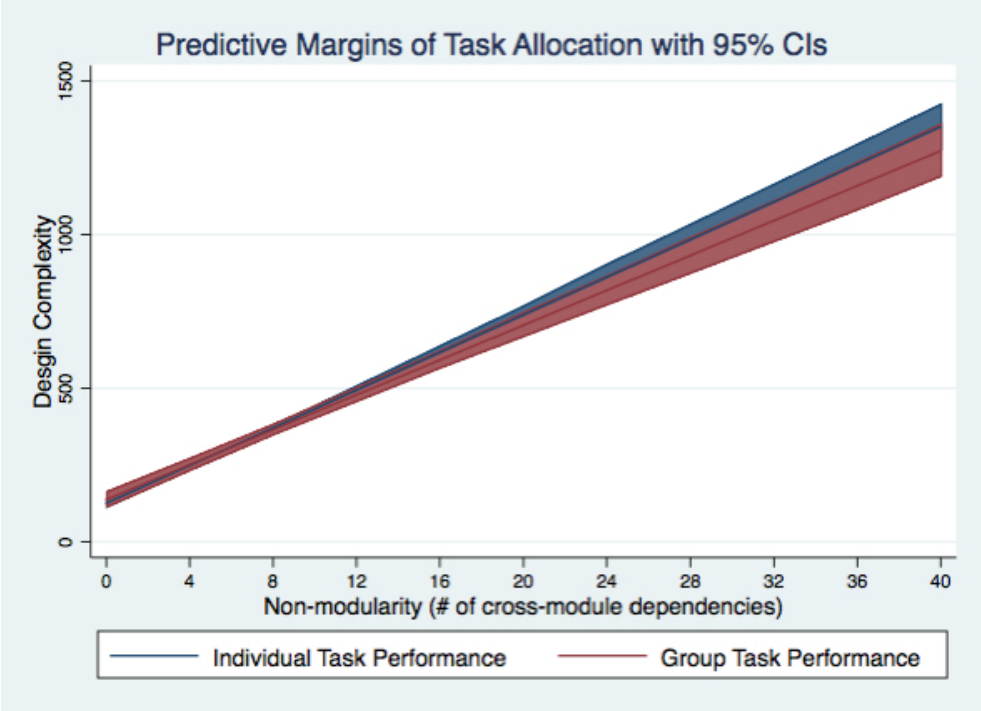


figure 6. Differential effect of non-hierarchy on design complexity (at the end of the development phase) as contingent upon task assignment decision

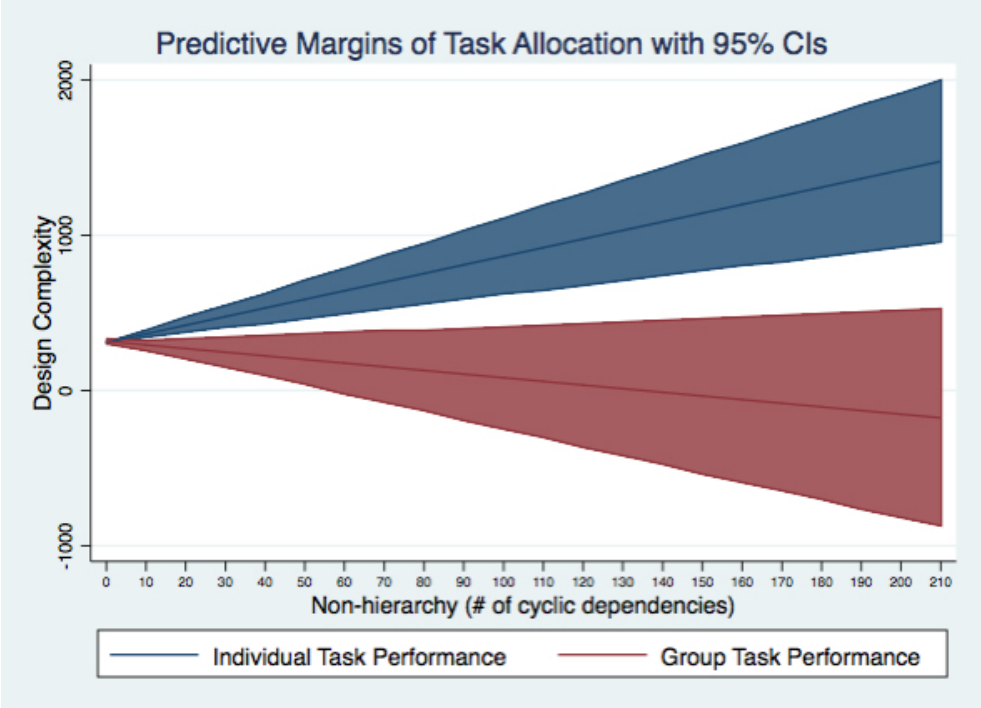


Table 1. Descriptive data of the analyzed software versions, measured at file level

Version	1.4	1.4.1	1.4.3	1.5	1.5.2	1.5.3	1.5.4	1.5.5	2.0.0	2.0.1	2.0.2	2.0.3	2.1.0
Release Dates	12/15/08	3/17/09	4/29/09	7/27/09	3/8/10	11/3/10	2/17/11	3/7/12	11/30/10	5/16/11	10/10/11	7/23/12	3/28/13
# of design elements (i.e. files)	2,840	2,841	2,841	3,136	3,296	3,304	3,308	3,444	3,502	3,568	3,609	3,632	3,736
# of *direct* technical dependencies b/w files	17274	17301	17306	18838	20032	20047	20061	20784	21067	21482	21774	22156	22914
Mean	6.242	6.256	6.261	6.218	6.282	6.28	6.272	6.239	6.18	6.173	6.189	6.265	6.321
Standard Deviation	26.64	26.66	26.67	26.6	26.86	26.86	26.84	26.74	26.21	26.3	26.39	26.76	27.81
Min	1	1	1	1	1	1	1	1	1	1	1	1	1
Max	566	566	566	596	614	614	614	616	626	633	637	655	674
# of *indirect* technical dependencies b/w files (Design Complexity)	241831	241656	241678	255015	264675	264676	264995	267733	282277	285365	288433	292772	301950
Mean	92.23	92.65	92.04	105.3	89.27	114.6	97.6	102.6	195.6	95.29	104.6	89.22	195.4
Standard Deviation	273.1	275.5	273.4	288.7	271.5	303.9	283.2	287.1	457.5	287.8	295	278.1	468.4
Min	1	1	1	1	1	1	1	1	1	1	1	1	1
Max	1,832	1,832	1,831	1,844	1,871	1,872	1,871	1,884	1,939	1,947	1,955	1,976	2,013
# of cross-module dependencies (Design Modularity)													
Mean	3.61	3.419	3.461	3.253	3.15	3.004	3.154	3.057	2.227	2.342	2.85	3.106	3.239
Standard Deviation	17.5	16.02	16.09	16.14	15.06	14.57	15.67	15.94	10.5	10.98	13.76	15.28	16.05
Min	0	0	0	0	0	0	0	0	0	0	0	0	0
Max	521	438	440	479	464	384	460	520	311	339	410	479	476
# of reciprocal (cyclic) dependencies (Design Hierarchy)	241831	241656	241678	255015	264675	264676	264995	267733	282277	285365	288433	292772	301950
Mean	0.91	0.875	0.907	1.063	1.127	1.767	0.977	1.034	13.04	0.927	1.024	0.929	12.46
Standard Deviation	3.413	3.202	3.382	3.629	3.895	8.002	3.54	3.67	49.43	3.387	3.665	3.27	48.29
Min	0	0	0	0	0	0	0	0	0	0	0	0	0
Max	23	23	24	22	25	58	27	24	209	26	24	23	210

Table 2. Descriptive statistics

Variable	Measure	Mean	Std. Dev.	Min	Max
Design Complexity	Total number of design elements that are directly or indirectly dependent on the focal element, measured at the end of the development phase of the given version	238.597	375.822	1	2015
Design Modularity	(Reversely coded) Number of dependent elements lying outside of the focal element's module	1.512	14.21	0	521
Design Hierarchy	(Reversely coded) Number of elements reciprocally dependent on the focal design element	3.823	24.81	0	210
Task Assignment	= 0 if the task on the focal element is performed by a single individual, = 1 if the task performed by a team of developers	0.086	0.28	0	1
Java Files	Identifies Java files (=1) from other file types (C/C++ files, coded as 0)	0.882	0.323	0	1
Cyclomatic Complexity	Internal complexity of the focal element (file), measured by mining its code lines.	5.942	8.991	0	127
Lines of Code	Number of lines of code contained in the focal element (file)	174.5	305.3	1	5500

Table 3. Pairwise correlation results

	1	2	3	4	5	6
1. Design Complexity	1.000					
2. Design Hierarchy (Reversely coded)	-0.330	1.000				
3. Design Modularity (Reversely coded)	0.030	0.050	1.000			
4. Java Files	0.030	0.030	0.040	1.000		
5. Cyclomatic Complexity	0.000	0.030	0.000	0.040	1.000	
6. Lines of Codes	0.000	0.040	0.020	-0.200	0.510	1.000

Correlations greater than 0.2 or smaller than -0.2 are significant, Number of Observations 29705

Table 4. OLS panel data regression results of design complexity$_t$, with random- and fixed-effects (H1 & H3)

Dependent Variable: Design Complexity $_t$	(1)	(2)	(3)	(4)
Non-Modularity$_{t-1}$	3.592499***	3.460649***	4.353783**	4.381252**
[# of cross-modular dependencies]	(.5929136)	(.5900099)	(1.354099)	(1.335767)
Non-Hierarchy $_{t-1}$	7.24202***	7.170846***	2.1396	2.101103
[# of cyclic dependencies]	(.6866302)	(.6865654)	(1.23238)	(1.231275)
Non-Modularity $_{t-1}$ ^{2}	-.0089444***	-.0091197***	-.0073909**	-.0074905**
	(.002169)	(.0020873)	(.0024877)	(.0024679)
Non-Hierarchy $_{t-1}$ ^{2}	-.0518911***	-.0519829***	-.0118143*	-.0117637*
	(.0032729)	(.0032639)	(.0057723)	(.0057667)
Non-Modularity $_{t-1}$ × Non-Hierarchy $_{t-1}$		.019196***		.0050742*
		(.0046088)		(.0023436)
Fixed effects (of design elements)	N	N	Y	Y
Constant	24.0801***	24.13905***	136.451***	136.314***
	(1.501979)	(1.501998)	(3.566419)	(3.571464)
R-squared	0.1453	0.1460	.4548398	.4548846
Number of observations	25746	25746	25746	25746

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

Table 5. Pooled regression results of design complexity$_t$, with the interaction effect of Individual (H2) and Team (H4) Task Assignments

Dependent Variable: Design Complexity $_t$	(1)	(2)	(3)	(4)	(5)	(6)	(7)
Non-Modularity$_{t-1}$	30.806***	31.070***	30.698***			30.939***	30.620***
[# of cross-modular dependencies]	(1.18)	(1.19)	(1.11)			(1.18)	(1.10)
Non-Hierarchy$_{t-1}$	5.216***			6.819***	6.120***	5.676***	5.555***
[# of cyclic dependencies]	(1.31)			(1.39)	(1.31)	(1.38)	(1.27)
Task Assignment$_{t'}$		4.829	17.440	14.751	13.973	12.602	22.001+
		(11.76)	(11.97)	(10.85)	(10.65)	(12.15)	(12.29)
Task Assignment$_{t'}$ X Non-Modularity$_{t-1}$		-1.035	-2.356***			-1.069	-2.247**
		(0.69)	(0.69)			(0.70)	(0.70)
Task Assignment$_{t'}$ X Non-Hierarchy$_{t-1}$				-12.382***	-8.442***	-10.980***	-7.905***
				(2.64)	(2.52)	(2.29)	(2.08)
Non-Modularity$_{t-1}$ ^{2}	-0.684***	-0.681***	-0.648***			-0.683***	-0.650***
	(0.04)	(0.04)	(0.04)			(0.04)	(0.04)
Non-Hierarchy$_{t-1}$ ^{2}	-0.023***			-0.030***	-0.026***	-0.025***	-0.024***
	(0.01)			(0.01)	(0.01)	(0.01)	(0.01)
Java Files$_{t-1}$	199.480***	189.910***	184.894***	341.161***	328.355***	200.654***	198.689***
	(5.71)	(5.36)	(7.66)	(5.35)	(8.16)	(5.80)	(7.89)
Cyclomatic Complexity$_{t-1}$	1.017***	1.090***	0.781***	3.154***	2.786***	1.049***	0.751***
	(0.15)	(0.16)	(0.16)	(0.16)	(0.16)	(0.16)	(0.16)
Lines of Code$_{t-1}$	0.190	-0.248	-0.947+	3.318***	3.218***	0.017	-0.870
	(0.50)	(0.54)	(0.57)	(0.64)	(0.67)	(0.54)	(0.57)
Constant	-0.045	11.411***	-11.972	-40.351***	-58.017*	-2.034	-13.188
	(3.61)	(2.65)	(12.43)	(4.17)	(25.63)	(3.81)	(13.23)
Versions' Fixed Effect	-	-	Y	-	Y	-	Y
Developers' Fixed Effects	-	-	Y	-	Y	-	Y
R-squared	0.388	0.384	0.479	0.296	0.394	0.388	0.483
Number of obs.	6675	6675	6675	6675	6675	6675	6675

*Robust standard errors in parentheses, *** p<0.001, ** p<0.01, * p<0.05, + p<0.1*

REFERENCE

Adner, R., & Levinthal, D. 2001. Demand heterogeneity and technology evolution: implications for product and process innovation. ***Management Science***, 47(5): 611–628

Alexander, C. 1964. Notes of the Synthesis of Form. ***Vasa***.

Baldwin, C., MacCormack, A., & Rusnak, J. 2014. Hidden structure: Using network methods to map system architecture. ***Research Policy***, 43(8): 1381–1397

Baldwin, C. Y., & Clark, K. B. 2000. ***Design rules: The power of modularity***, vol. 1. The MIT Press

Baldwin, C. Y., & Clark, K. B. 2006. Modularity in the design of complex engineering systems. ***Complex engineered systems***: 175–205. Berlin Heidelberg: Springer

Banker, R. D., Davis, G. B., & Slaughter, S. A. 1998. Software Development Practices, Software Complexity, and Software Maintenance Performance: A Field Study. ***Management Science***, 44(4): 433–450

Betz, S., Fricker, S., Moss, A., Afzal, W., Svahnberg, M., et al. 2013. An Evolutionary Perspective on Socio-Technical Congruence: The Rubber Band Effect. ***Replication in Empirical Software Engineering Research (RESER), 2013 3rd International Workshop on***, 15–24. IEEE

Blondel, V. D., Guillaume, J.L., Lambiotte, R., & Lefebvre, E. 2008. Fast unfolding of communities in large networks. ***Journal of Statistical Mechanics: Theory and Experiment***, 2008(10): 6

Browning, T. R. 2001. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. ***IEEE Transactions on Engineering Management***, 48(3): 292–306

Brusoni, S., & Prencipe, A. 2001. Managing Knowledge in Loosely Coupled Networks: Exploring The Links Between Product and Knowledge Dynamics. ***Journal of Management Studies***, 38(7): 1019–1035

Brusoni, S., & Prencipe, A. 2011. Patterns of modularization: The dynamics of product architecture in complex systems. ***European Management Review***, 8: 67–80

Cabigiosu, A., & Camuffo, A. 2016. Measuring Modularity: Engineering and Management Effects of Different Approaches. ***IEEE Transactions on Engineering Management***, PP(99): 1–12

Cataldo, M., Herbsleb, J., & Carley, K. 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. In ***Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement,*** PP:2-11

Colfer, L., & Baldwin, C. 2016. The Mirroring Hypothesis: Theory, Evidence and Exceptions. ***Industrial and Corporate Change***, 25(5): 709–738

Conway, M. E. 1968. How do committees invent? ***Datamation***

David, P. A. 1985. Clio and the Economics of QWERTY. ***The American Economic Review***, 75(2): 332–337

Eppinger, S. D., & Browning, T. R. 2012. ***Design structure matrix methods and applications***. MIT press.

Ethiraj, S. K., & Levinthal, D. 2004. Bounded rationality and the search for organizational architecture: An evolutionary perspective on the design of organizations and their evolvability. ***Administrative Science Quarterly***, 49(3): 404–437

Ethiraj, S. K., & Levinthal, D. A. 2002. Search for architecture in complex worlds: an evolutionary perspective on modularity and the emergence of dominant designs. ***Wharton School, University of Pennsylvania***

Ethiraj, S. K., & Posen, H. E. 2013. Do Product Architectures Affect Innovation Productivity in Complex Product Ecosystems? In R. Adner, J. E. Oxley, & B. S. Silverman (Eds.), ***Collaboration and Competition in Business Ecosystems***, vol. 30: 127–166. Emerald Group Publishing Limited

Ethiraj, S. K., Ramasubbu, N., & Krishnan, M. S. 2012. Does complexity deter customer-focus? ***Strategic Management Journal***, 33(2): 137–161

Furlan, A., Cabigiosu, A., & Camuffo, A. 2014. When the mirror gets misted up: Modularity and technological change. ***Strategic Management Journal***, 35(6): 789–807

Henderson, R., & Clark, K. 1990. Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. ***Administrative Science Quarterly***, 35(1): 9–30

Herbsleb, J. D. 2007. Global software engineering: The future of socio-technical coordination. ***2007 Future of Software Engineering***, 188–198. IEEE Computer Society

Kwan, I., Cataldo, M., & Damian, D. 2012. Conway's Law Revisited: The Evidence For a Task-based Perspective. ***IEEE Software***, 29(1): 1–4

MacCormack, A., Baldwin, C., & Rusnak, J. 2012. Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis. ***Research Policy***, 41(8): 1309–1324

Maccormack, A., Baldwin, C. Y., & Rusnak, J. 2010. The Architecture of Complex Systems : Do Core-periphery Structures Dominate? ***Academy of Management Proceedings***.

MacCormack, A., Rusnak, J., & Baldwin, C. 2007. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. ***Harvard Business School Working Paper, 08-038***

MacCormack, A., Rusnak, J., & Baldwin, C. Y. 2006. Exploring the Structure of Complex

Software Designs: An Empirical Study of Open Source and Proprietary Code. ***Management Science***, 52(7): 1015–1030

Murmann, J. P., Frenken, K., & Peter, J. 2006. Toward a systematic framework for research on dominant designs, technological innovations, and industrial change. ***Research Policy***, 35(7): 925–952

Parnas, D. L. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. ***Communications of the ACM***, 15(12): 1053–1058

Ramasubbu, N., & Kemerer, C. 2014. Managing Technical Debt in Enterprise Software Packages. ***IEEE Transactions on Software Engineering***, 40(8): 758–772

Ramasubbu, N., & Kemerer, C. F. 2015. Technical Debt and the Reliability of Enterprise Software Systems: A Competing Risks Analysis. ***Management Science***, 1–48

Rivkin, J. W., & Siggelkow, N. 2003. Balancing Search and Stability: Interdependencies Among Elements of Organizational Design. ***Management Science***, 49(3): 290–311

Sanchez, R., & Mahoney, J. T. 1996. Modularity, flexibility and knowledge management in product and organization design. ***Strategic Management Journal***, 17(SI): 63–76

Simon, H. 1962. The architecture of complexity. ***Proceedings of the American Philosophical Society***, 106(6): 467–482

Smith, R. P., & Eppinger, S. D. 1997. Identifying Controlling Features of Engineering Design Iteration. ***Management Science***, 43(3): 276–293

Srikanth, K., & Puranam, P. 2011. Integrating distributed work: Comparing task design, communication, and tacit coordination mechanisms. ***Strategic Management Journal***, 32(February): 849–875

Srikanth, K., & Puranam, P. 2014. The Firm as a Coordination System: Evidence from Software Services Offshoring. ***Organization Science***, (February).

Sturtevant, D., & MacCormack, A. 2013. The Impact of System Design on Developer Productivity. ***Academy of Management Conference. Orlando (USA)***

Sturtevant, D., MacCormack, A., Magee, C., & Baldwin, C. 2013. ***Technical Debt in Large Systems : Understanding the cost of software complexity***. MIT

Thompson, J. D. 1967. ***Organizations in action: Social science bases of administrative theory***, vol. 1. Transaction Publishers

Valetto, G., Helander, M., Ehrlich, K., Chulani, S., Wegman, M., et al. 2007. Using software repositories to investigate socio-technical congruence in development projects. ***Proceedings of the Fourth International Workshop on Mining Software Repositories***, 25. IEEE Computer Society

von Hippel, E. 1990. Task partitioning: An innovation process variable. ***Research Policy***, 19(5): 407–418

Zhou, Y. 2013. Designing for complexity: using divisions and hierarchy to manage complex tasks. ***Organization Science***, (October).

ESSAY 2

Why Product and Organizational Architectures Misalign? A Study of the Microdynamics of the "Mirroring Hypothesis"

Mahdi Ebrahim
Department of Management and Technology,
Bocconi University
mahdi.ebrahim@phd.unibocconi.it

Arnaldo Camuffo
Department of Management and Technology,
Bocconi University
arnaldo.camuffo@unibocconi.it

ABSTRACT

Product architectures should reflect the organizational structures that design them (the "mirroring hypothesis"). If they do not, modularity literature has documented a number of potential negative performance effects. We study how and why product and organizational architectures do not mirror, analyzing the micro-dynamics and determinants of product cross-module dependencies for which there is no corresponding organizational tie. Building on complexity theory applied to technological and social systems, we: a) hypothesizes that non-mirroring is a function of the nature of the product development tasks, the proximity of developers, and time pressure; and, b) test these hypotheses on a unique micro-level dataset, documenting the development of an enterprise software over a course of 60 months and along with 13 versions. Our estimates, robust to different model specifications and identification strategies, suggest that non-mirroring between product and organizational architectures increases when development tasks are routine, engineers are not co-located, and over time as the deadline of releasing new software version approaches. Our findings contribute to the organizational design literature, documenting how micro-organizational choices give rise to the misalignment between product and organizational architectures and offers engineers insights into how to prevent the negative performance outcomes that stem from non-mirroring.

Keywords: Modularity, Mirroring hypothesis, Interdependencies, Software Industry

INTRODUCTION

Over the past two decades, a wide range of studies located at the intersection of innovation, strategy, and organization theory, has investigated the ways in which the architectures of complex technological and social systems co-vary (e.g., Baldwin & Clark, 2000; Henderson & Clark, 1990; Sanchez & Mahoney, 1996). This field of inquiry has been recently systematized under the conceptual framework of the "mirroring hypothesis," which poses that a technological system and the organizational structure that designs it should share similar architectural properties, for example in terms of degree of hierarchy and near-decomposability (Colfer & Baldwin, 2016). This topic is core in the current management debate as it addresses one of the critical challenges that scholars and practitioners face: the simultaneous design and management of multiple complex technological and social systems.

Ethiraj and Levinthal (2004) summarize this challenge referring to Simon's (1962) foundational work on "bounded rationality" and the "architecture of complexity": given that modularity is a desirable attribute of a complex system (a product, a technology, as well as an organization), how do bounded-rational engineers/managers identify and uncover structures of modularity in order to design complex technological and social systems that remain evolvable. Reformulated under the "mirroring hypothesis" conceptual framework, the research question becomes whether or not bounded-rational managers and engineers can concurrently design technological and social systems that are modular (and therefore evolvable), regardless of the complexity of the purpose for which such systems are designed. If, as suggested by the "mirroring hypothesis", the architectural properties of such systems should be similar and evolve correspondingly, what determines the cases in which they do not mirror?

Many empirical studies support the "mirroring hypotheses" and implicitly or explicitly suggest that there are significant negative performance implications when it comes to architectural mismatches ("non-mirroring") between products and organizations. Other studies, however, suggest that non-mirroring between the degree of modularity of products and organizations not only is possible, but also widespread, for reasons related to strategic behaviors (Fixson & Park, 2008) or to epistemic interdependence between actors (Furlan et al., 2014). These studies, however, are static in nature and take a macro perspective. Furthermore, they do not investigate why "non-mirroring," being sub-optimal, occurs and how it evolves over time. This study tackles these unexplored issues. It focuses on "non-mirroring" product architectures and organizational structures and tries to understand its determinants. It does so at the micro-level, using a unique dataset from an industrial software provider. We analyze 13 versions of a commercial software package over the course of 60 months. We focus on cross-module technical dependencies for which there is no corresponding organizational tie. Then, building on complexity theory applied to product and organization design, we:

a) describe the dynamics of cross-module interfaces and of the corresponding organizational interdependencies, showing that product architectures and organizational structures reciprocally influence each other over time, and posing that, for a given product organization, non-mirroring increases over time across versions of the product, leading to inter-generational complexity growth.

b) hypothesize three micro-level determinants of "non-mirroring," (narrowly defined, in our case, as the absence of organizational dependencies (cross-developer collaborations) in presence of cross-module interdependencies: i) the degree of innovativeness of development tasks; ii) the developers' physical proximity; and iii) time pressure.

Applying a variety of regression techniques, we test our hypotheses and find that "non-mirroring" is more likely:

i) when the nature of the development task is incremental changes (i.e. modifying existing features and/or fixing functional defections), while it is less likely when the nature of the development task is the creation of new product features;

ii) when dependent developers are distant from one another;

iii) as the release date of the new version of the product approaches.

Our findings shed light on the structure, dynamics, and determinants of "non-mirroring," and contribute to the modularity debate by introducing a co-evolutionary approach to the understanding of the relationship between product architectures and organizational structures and their misalignment.

The paper is organized as follows: Section two reviews modularity literature and the studies on the "mirroring hypothesis". Section three conveys the arguments and develops the hypotheses. Section four explains the research data and methodology. Section five reports the results while section six discusses the findings and draws some theoretical and managerial implications, highlighting the study limitations and some directions for future research.

THEORETICAL BACKGROUND

Definition of Modularity

Near-decomposability, or modularity is an architectural property that consists in a system arrangement in which subsystems have more interdependencies within them (tight coupling) than between them (loose coupling) (Alexander, 1964; Baldwin & Clark, 2000; Parnas, 1972), which represents a key theme in strategy, innovation, and organization research

(Schilling, 2003; Orton & Weick, 1990). Modularity is typically taken to mean that the interdependence between subsystems is sufficiently low to permit separability and recombinability. When two product subsystems are independent from each other (high degree of modularity), changes in one of them do not affect the functioning of the other. Technical dependencies among modules (subsystems) are minimized, standardized, and stabilized so that the subsystems are basically separable and recombinable. Thanks to the minimized and standardized cross-module interfaces, the subsystems perform appropriately together and can be modified at little cost of propagating changes to unintended modules (Baldwin & Clark, 2000; MacCormack et al., 2006).

On the organizational side, the individuals, teams, or suppliers that develop the technological subsystems (e.g. product components) do not need to interact and share information with each other (or need to do so minimally), since relevant knowledge is encapsulated in the modules or already efficiently contained in the cross-module interfaces (information hiding (Parnas, 1972)). Conversely, if two product subsystems are dependent on each other (low degree of modularity), changes in one of them affect the functioning of the other. Technical dependencies among subsystems are numerous and "thick" so that the subsystems are not separable. The individuals, teams, or suppliers that develop them must frequently interact and intensely communicate with one another in order to share the necessary information about how the components work together. They will try to design the interfaces optimally, and streamline, standardize, and stabilize them so that the product components perform appropriately together (Baldwin & Clark, 2000; Colfer & Baldwin, 2016). Nonetheless, the design and development process of dependent components will require a lot of rework and iterations of design efforts, for example, to solve unforeseen problems and make sure that the components function well together (MacCormack et al., 2012). Typically, cross-

component interface optimization via modularization requires substantial effort, learning and problem solving (Colfer & Baldwin, 2016; MacCormack et al., 2006), which only possibly and eventually might result in modular product designs (MacDuffie, 2013).

The application of modularity to management studies draws heavily on Herbert Simon's work on the architecture of complexity, and, more specifically, on the concept of hierarchically nested, nearly-decomposable systems (1962). Simon noted that many complex systems are actually composed of component subsystems that are relatively independent. The independence of subsystems, or more precisely, that subsystems will have more interdependencies within them than between them is generally considered as a desirable attribute and a system design principle (Schilling, 2003). In the management literature, the concept of modularity is used to analyze products and organizations as complex, hierarchically-nested technological systems that comprise a large number of subsystems (components) with many interactions among them (Henderson and Clark, 1990). Capturing the way in which a product design is decomposed into different parts or modules, helps to conceptually characterize different product architectures, i.e. the scheme by which a product's functions are allocated to its components (Ulrich, 1995).

Modularity is therefore typically taken to mean that the interdependence between subsystems is sufficiently low to permit separability, i.e. that either the subsystems themselves can be physically separated, or the processes pertaining to them can be separated, and so, management and development of subsystems can be performed independently (Cabigiosu & Camuffo, 2016).

Murmann and Frenken (2006) relate the concept of separability to the concept of "pleiotropy." The pleiotropy of a subsystem (product component) is the number of functions affected by it, or the "number of service characteristics that will change their value when this

component in the system is changed" (p. 941). High-pleiotropy components affect many things the product does, while low-pleiotropy components affect only a few. In other words, separability allows local changes (within the subsystems/modules) with no or little propagation of changes at the system level. This reduces the complexity of development processes (e.g., permitting modular or incremental innovations (Henderson & Clark, 1990) and the complexity of managing the production processes). Therefore, production units or external suppliers can specialize on more narrowly defined components (Fine, 1998; Sanchez & Mahoney, 1996).

A second key aspect in the engineering and management uses of the concept of modularity (but that is not always invoked in other research domains) is recombinability. Recombinability refers to a degree of flexibility in the combination and configuration of subsystems. For example, it may be possible to substitute one component for another, to add or subtract components, or to alter the way in which subsystems are combined (Mikkola, 2006). The recombinability of modular products means that heterogeneous components can be mixed and matched into products and services to meet the heterogeneous demands of customers (Schilling, 2000). Substitutability (replacing one component with another), expandability (adding on additional components), and upgradeability (replacing a component with a version that is newer, bigger, etc.) are all types of recombinability (Salvador, 2007).

The specificity of the functions implemented by a subsystem is a key property that determines, in part, the degree to which that subsystem can be separated from the rest of the product and managed independently. If subsystems share some functions, it would be difficult to isolate them, for example, during the design and production phases. Functional isolation of components, however, is not a sufficient condition to define product modularity. The re-combinability of a product's components depends on component interfaces. Rather than having

each component designed to work specifically with other particular components in a tightly coupled system, a modular product design utilizes open standard (or closed standard) interfaces that permit a range of components to be recombined and to function and interact without undesired or uncontrolled effects. Standard interfaces fix the coupling rules among components (Sosa, Eppinger, & Rowles, 2003), and thus allow firms to know a priori how components will interact[i]. In this study, we will use the technical definition of modularity that, building on Baldwin and Clark's definition (1997; 2000) focuses on technical dependencies among components or interfaces.

This approach assumes that the number/type of interfaces is a good proxy for the degree of modularity, as it indirectly captures also the degree of functional isolation[ii]. Most studies of product modularization adopt this definition (Guo & Gershenson, 2003; Huang & Kusiak, 1998; Larses & Blackenfelt, 2003; Martin & Ishii, 2002; Pimmler & Eppinger, 1994; Sosa et al., 2003; Whitfield, Smith, & Duffy, 2002) and analyze cross-component interfaces using a specific analytic tool, the Design Structure Matrix (DSM) (Browning, 2001; Eppinger & Browning, 2012), and its developments (Sosa et al., 2003; Sturtevant & MacCormack, 2013). This matrix describes (contingent on the version used) the number and nature of cross-component interfaces lending insights into the degree of coupling between components as well as their distribution.

The same concept underlies also Hsuan (1999) and Momme et al (2000) definitions of "modularity" as clustering the within-component dependencies and standardizing across-component interfaces. Such definition of modularity was pioneered in the engineering and management literature during the late '80s, soon becoming prevalent in the strategic management literature (Garud & Kumaraswamy, 1993).

The Mirroring Hypothesis: Theoretical Underpinnings and Empirical Evidence

The "mirroring hypothesis" posits that a technological system and the organizational system that designs it should have similar architectural properties, such as the degree of modularity, and that these architectural properties should co-vary (Henderson and Clark, 1990; Sanchez and Mahoney, 1996; Baldwin and Clark, 2001; von Hippel, 1990). When two product components are dependent on each other, any change in one of them during the development process affects the functioning of the other dependent component. Therefore, during the development process, two developing teams designing two distinct components should communicate with one another to share the necessary information about the interdependencies between the two components, while at the same time try to simplify and standardize the relevant interfaces (Brusoni & Prencipe, 2011; Stefano Brusoni & Prencipe, 2001; MacCormack, Baldwin, & Rusnak, 2012; MacCormack, Rusnak, & Baldwin, 2006; Sanchez & Mahoney, 1996). Moreover, the design and development process of dependent components typically requires a significant amount of effort, several design iterations and possibly a sizable amount of rework, in order to make sure that the two components function well in tandem (MacCormack et al., 2012). Abundant empirical research documents and supports such theoretical argument (Colfer & Baldwin, 2016; MacCormack et al., 2006). A similar intuitive proposition is also suggested by studies in software engineering literature, where it is referred to as "Conway's law" (Conway, 1968; Kwan, Cataldo, & Damian, 2012), and, more recently, "socio-technical congruence" (Betz et al., 2013; Cataldo, Herbsleb, & Carley, 2008; Herbsleb, 2007; Valetto et al., 2007).

While "mirroring" is theoretically appealing and widely empirically supported, other studies document widespread "non-mirroring" between product and organizational architectures and investigate under which conditions the "mirroring hypothesis" might not hold.

Some of them suggest that mirroring is not necessary, as firms might strategize and use non-mirroring as a competitive weapon, or have heterogeneous capabilities that help turning non-mirroring into their competitive advantage (Cabigiosu, & Camuffo, 2012, 2016; Fixson & Park, 2008). Others question the performance effects of mirroring (i.e. the idea that non-mirroring implies worse performance) and point to environmental, industrial, and firm-level contingencies that might suggest the contrary (Srikanth & Puranam, 2011). Others suggest that mirroring might be a special case, the unstable output of learning and search processes, and that product and organizational architectures typically do not mirror but occasionally, e.g. when technology is stable (Furlan, Cabigiosu, & Camuffo, 2014). Overall, these studies implicitly or explicitly maintain that non-mirroring is at least as frequent as mirroring and that the mismatch between how technical dependencies are clustered within and among product components versus how organizational dependencies are grouped within and across developers, jobs, teams and organizational units is widespread in many organizations (Colfer & Baldwin, 2016).

Reviewing the studies on the mirroring hypothesis, Colfer and Baldwin (2016) observe and classify two types of "non-mirroring," i.e. two types of mismatch between component interdependencies (usually in the form of development task dependencies) and organizational ties (such as firm co-membership, developers' proximity, formal or informal communication, information sharing and collaboration) (see figure 1 below).

---

insert figure 1 about here.

The first type of mismatch represents cases where cross-boundary organizational ties—such as cross-developer, cross-team, cross-unit or cross-firm dependencies—exist, even in

presence of few, thin and standardized cross-module technical dependencies (Brusoni & Prencipe, 2011; Brusoni, Prencipe, & Pavitt, 2001). Specifically, this definition corresponds to cases in which the boundaries of a job or unit include tasks that cut across different product modules, so that individuals or teams operate, interact, and collaborate on multiple modules despite the few, thin and standardized technical interdependencies among them. For example, Furlan et al. (2014) document cases of persisting communication and collaboration between buyers and suppliers, occurring even when the relevant product components have no technical dependencies. More generally, this mismatch corresponds to situations where "tight-knit teams within single firms successfully created and capitalized on modular product architectures that did not mirror their own organizations" (Colfer and Baldwin, 2016, p. 723).

The second type of mismatch regards cases where "diverse parties with few organizational ties work together […] on a complex, technically interdependent product or system" (Colfer and Baldwin, 2016, p. 725). In other words, individuals or teams contribute to develop highly interdependent product modules without communicating and sharing information with the corresponding cohorts and restraining from collaboration while working on the dependent modules. Hence, at a more micro level, technical dependencies spanning across product modules are addressed by independent and secluded individuals or teams (Colfer & Baldwin, 2016; Srikanth & Puranam, 2011).

The two types of mismatch identified in Figure 1 are helpful to classify different cases of "non-mirroring" at a given point in time. However, they do not help us understand either their dynamics or their determinants. This study takes on this research challenge. It focuses on one type of non-mirroring, mismatch type II, and investigates how it might evolve over time and what determines it. We focus on this type of "non-mirroring" as the extant literature about

mismatch type II is comparatively more scant, even if it documents the unfavorable effects of "non-mirroring" on product quality, and more generally, on firms' performance (Gokpinar, Hopp, & Iravani, 2010; Sosa, Eppinger, & Rowles, 2004).

Colfer and Baldwin (2016) summarize the studies that indirectly and directly test the mirroring hypothesis and provide a comprehensive overview of the relevant research. They acknowledge that conventional views on mirroring tend to be simplistic and general, and emphasize the previous calls for a more nuanced theory of mirroring, as suggested by empirical studies spanning a range of different industries (Brusoni & Prencipe, 2001; Hoetker, 2006; Staudenmayer, Tripsas, & Tucci, 2005). Their review also suggests that a more contingent view of the mirroring hypothesis is necessary, in order to highlight industry- and firm-level moderators influencing where mirroring holds or breaks. Furthermore, other recent studies call for exploring the intertemporal dynamics of the mirroring hypothesis, how it unfolds over the lifecycle of industries, organizations, and products, and what determines it.

Such calls, however, have gone largely unheeded, as existing research almost always adopts a static approach to test the mirroring hypothesis, focusing on the correspondence between product architecture and organizational structure at a given point in time. Some studies offer insights about the dynamics and determinants of "non-mirroring," but, due to the limited data available, remain cross-sectional (MacCormack et al., 2012). Others offer a partly dynamic perspective, in the sense that technical dependencies and organizational ties are studied at different point in times (but still, with cross-sectional data) (Cabigiosu & Camuffo, 2012; Furlan et al., 2014), or that technical dependencies' evolution is studied over time, but given a certain organizational structure (Baldwin, MacCormack, & Rusnak, 2014; MacCormack et al., 2012).

This study departs from the prevailing approach and tries to understand "non-mirroring" studying how product architectures and organizational structures co-evolve over time. It adopts a processual view and examines the micro-processes through which various product designs (software versions in our setting) are designed and released over time by an evolving organization. Our data allow to observe not only various product designs (software versions) over time as the output of development processes, but also the corresponding organizational structures in the form of task-partitioning, dependencies and collaboration patterns between developers.

HYPOTHESES

Task Innovativeness

Product development comprises different types of tasks. Some regard the fine tuning of existing products, their updating, the solution of problems deriving from quality issues or other types of issues emerging from customers' feedback. These activities typically represent incremental changes to the product design and the related problem solving activities typically requires the use and recombination of existing knowledge. Other activities instead involve creating new features and incorporating new functionalities to a given product design.

Creating new product features essentially means to incorporate new technologies in the product by generating new knowledge—a process replete with continuous experimentation and exploration. Considering the uncertain, chaotic, and possibly haphazard nature of this exploratory process, the architectural complexity of the product design resulting from such process might unintendedly generate "non-mirroring," especially if the outcome of the development process is not properly designed and managed. Adding new product features that accommodate new technologies and meet customers demand is a creative task that requires

high level of individual and organizational attention (Ocasio, 1997, 2010), thus, rendering non-mirroring less likely to take place vis-à-vis routine tasks of modifying existing product features. In fact, the development work of generating new product functionalities, which affect internal and external design elements, and thus, generally involves intentionality, is more likely to be carefully planned and actively supervised, driving higher diligence, attention, and engagement by engineers.

Similarly, the adoption of agile development practices—such as scrum, sprints, morning standup meetings and abundant ad-hoc communications between developers who work on the same files or dependencies—are also likely to be more effectively used in the case of development tasks regarding new product features (Dybå & Dingsøyr, 2008; Pikkarainen, Haikara, Salo, Abrahamsson, & Still, 2008), thus reducing the probability that, if cross-modular links exist, the corresponding organizational ties are activated.

Overall, it is reasonable to expect that, other things being equal, non-mirroring is less frequent when engineers work on new product features, generating new bodies of knowledge and embedding them into the existing architecture of complex products.

Conversely, developing and maintaining a complex product also includes attempts to modify, reuse, and reconfigure existing features of the product by refining and recombining bodies of knowledge already embedded in the product architecture. These tasks involve local search over the existing product's architecture in order to find optimal and satisfying solutions to surfaced problems (Karim & Kaul, 2015; Rivkin & Siggelkow, 2003; Siggelkow & Levinthal, 2003).

These tasks generally include incremental changes to existing product features, either inherited from the previous product versions or created earlier during the development of the

current version. Performing these tasks, engineers solve problems locally, searching for close by solutions, relying on their existing knowledge of the product components' interdependencies, leveraging on their current competencies, habits, and routines. This might result in myopic search behavior, and translate in suboptimal solutions (Cyert & March, 1963; Levinthal & March, 1993, 1981; Levitt & March, 1988) thus potentially leading to neglect the activation of organizational ties corresponding to existing cross-module technical dependencies.

Making changes to a specific part of a given product design might lead engineers to stay overly focused on the specific dependencies and interfaces. Such focus might inhibit their ability to identify and foresee the propagation of the design changes they implement. Hence, engineers might not accurately identify and account for the cross-module interdependencies, and, consequently, fail to activate the relevant organizational ties and engage in the necessary collaborations. Thus, we expect to observe comparatively more cases of overlooked organizational dependencies during routine development tasks.

Overall, we hypothesize that non-mirroring (of the type we analyze) is more likely to take place in the case of development tasks that require the modification of current product features, compared with the case of tasks that require the creation of new features. Hypothesis 1 follows:

> *Hypothesis 1. The occurrence of non-mirroring is more likely when performing routine development tasks (modification of existing product features), compared to performing innovative tasks (creation of new product features).*

Physical Proximity

Existing organization literature argues that physical proximity of employees facilitates communication and collaboration. Physical proximity simplifies relationships, reduces search

costs, and favors the establishment of ties with relevant co-workers. In addition, by providing more opportunities to meet, as well as other forms of informal communication, physical proximity helps to create a shared understanding of the activities. In the case of product design, co-location (for example in large *obeya rooms*) allows a common understanding of the development process, of the development tasks, and of the product architecture, which in turn, facilitates task allocation to engineers and coordination (Srikanth & Puranam, 2011, 2014; Hao, Feng, & Frigant, 2015).

Proximity is particularly beneficial in coordinating development tasks conducted by engineers working on a given module. The proximity of engineers working on specific modules help them stay focused on within-module technical dependencies and minimize unnecessary cross-developer (or cross-team) organizational dependencies (such as coordination, communication, and collaboration). At the same time, proximity also facilitates module boundary spanning, making it easier, for engineers working on separate module and in different teams, to identify relevant counterparts working on other modules in different teams, to consider, establish, change or improve cross-module interfaces (technical dependences), and reach out to other engineers to coordinate and collaborate, should this be necessary.

Performing tasks on cross-module dependent components is particularly challenging, since the engineer or team assigned to a given module might not be knowledgeable about the components belonging to other modules. Therefore, in order to coordinate, engineers have either to take a lot of time and energy to learn about the content and embedded knowledge in other modules, their components and their functionalities, or refer to engineers belonging to teams in charge of the design of the other modules, asking for the information necessary to perform flawlessly the given cross-module task. In both the cases, physical proximity facilitates

coordination (learning in the former case, information exchange in the latter) reducing the information cost and cognitive overburden required to module boundary spanning (Hoegl & Proserpio, 2004; Kraut & Streeter, 1995).

Hypothesis 2 follows:

Hypothesis 2: The occurrence of non-mirroring is less likely when engineers are co-located.

Time Pressure

The development of a new product requires a lot of exploration and experimentation (Murmann & Frenken, 2006; Suárez & Utterback, 1995), with engineers vigorously exchanging information about how to solve design problems. Design options (solutions of problems) are defined, discussed, agreed upon, and eventually implemented through extensive collaboration among all engineers, no matter the extent to which the components of the product design are technically dependent. In knowledge-intensive contexts, managing such innovative activities is even more challenging as problem solving requires to generate and share knowledge among engineers working on different product modules, so that interferences and confounding changes in design decisions are avoided.

However, as deadlines to release new products approach and time pressure mounts, developers' productivity and quality tend to decrease. A wide stream of organizational behavior research demonstrates that time pressure has negative effect on performance because "it places constraints on the capacity for thought and action that limit exploration and increase reliance on the well-learned or heuristic strategies" (Moore & Tenney, 2012, p. 305) leading to inferior performance. Time pressure might harm engineers' performance even stronger in the presence

of delay penalties, budget constraints or strong incentives to meet release deadlines. In this case engineers might opt to collect less information than necessary while choosing a problem-solving strategy and make decisions more quickly (Christensen-Szalanski, 1980), relying to their initial impressions (Heaton & Kruglanski, 1991) and cognitive heuristics (Kruglanski & Freund, 1983). More specifically, engineers that are afraid of the negative consequences of missing deadlines might take "shortcuts" while performing the assigned tasks (Austin, 2001).

These shortcuts might take the form of not interacting, communicating and collaborating with engineers working on other product modules, linked to the focal one through cross-modular links (interfaces). Under time pressure, engineers might fail to understand which components will be affected by the design changes they make, or lack the time, energy and resources to do so (Baumann & Siggelkow, 2013; Ethiraj & Levinthal, 2004; Simon, 1972). Further, engineers will have little time and cognitive slack to exchange the necessary information and, for example, adequately participating to meetings, conference calls, or using collaboration technologies (Moore & Tenney, 2012). Therefore, we expect to observe more instances of overlooked organizational dependencies later during the development phase as release deadlines approach. Thus, hypothesis 3 follows:

> *Hypothesis 3: the occurrence of non-mirroring is more likely as the deadline for product release approaches.*

DATA AND METHODS

Research Setting and Design

We use a unique dataset of micro-level information on software architectures and engineers' collaboration patterns in software development. The dataset spans 60 months of development of 13 versions of a commercial software system, collected from the software

division of an HVAC (Heating, ventilation and air conditioning) technology premium supplier. The software studied is a plant supervision package—an integrated control solution that improves energy savings in HVAC systems. In 2015, the company reported revenues of 250 million euros with a 15% EBITDA margin, and is planning to go public soon.

The software division started the systematic recording of development data (software architecture, development tasks, etc.) in early 2008, when the development team was working on version 1.4 of the software. Since then the data have been stored in the company's digital repository. No data or incomplete data are available for earlier versions, also because, as confirmed by the chief software engineer in the interviews we conducted, version 1.4 of the analyzed software represented the first real full version of the software and a significant discontinuity in terms of software architecture and development activities compared with previous versions. Since version 1.4, two incremental versions of the software were released internally (1.4.1 and 1.4.3), before the market release of version 1.5. After this, in response to the competitors' launch of new software applications with more user-friendly interfaces, the company decided to branch a new generation of the software with new functionalities (2.X branch, including 2.0.0, 2.0.1, 2.0.2, 2.0.3, and 2.1.0), while at the same time continuing to develop new versions of the software on the existing branch (1.X branch, including 1.5.2, 1.5.3, 1.5.4, and 1.5.5). These new versions basically included improvements in the form of service packs (figure 2). Therefore, our dataset is divided into two parts, each capturing the data of one branch. The first dataset includes versions: 1.4.0, 1.4.1, 1.4.3, 1.5, 1.5.2, 1.5.3, 1.5.4, and 1.5.5 (called branch 1.X henceforth), while the second dataset includes data of versions: 1.4.0, 1.4.1, 1.4.3, 1.5, 2.0.0, 2.0.1, 2.0.2, 2.0.3, 2.1.0 (called branch 2.X henceforth). We pooled the data of both branches into a single database, and thus the regression results reported in the finding section pertain to both branches. Nonetheless, separate analyses of the two branches

demonstrate similar patterns and results. The following table reports a description of each version's characteristics.

Insert figure 2 about here.

Insert table 1 about here.

Software development is a particularly suitable setting to address our research question (how "non-mirroring" evolves over time and what drives it) since it allows to collect detailed and objective micro-data about product components (i.e. software files) and their technical dependencies (the product architecture), as well as data about the organization developing the software (the organizational architecture), including information about task allocation (jobs, teams, individuals), the characteristics of the tasks accomplished during software development and maintenance, and the organizational dependences among software developers.

In this context, to accomplish a task—either meant to develop a new feature or to modify an existing feature—developers identify the files that must be changed, downloads a copy of them from the software repository, edits/revises them, and eventually uploads them again to the repository. This process is called a "revision." Therefore, a given "revision" involves a set of "tasks" performed on a few files. Using the company's record of "revisions" we collected different sorts of data pertaining to each of the "tasks" performed during the development of the studied software. Furthermore, the availability of micro-level data of software architectures—all the way to the single-file level—allows to detect the cross-file and cross-directory dependencies, and measure the architectural properties of the files, enabling us to

examine the co-changes of the product architecture and organizational structure at the most micro level.

Data Description

We received access to different data repositories of the company, then merged, and organized them into a comprehensive dataset, including detailed information about files, their dependencies, tasks, developers, and collaborations for all versions of the analyzed software. More specifically, the dataset includes the following technical data about the software versions:

a. Files' technical characteristics and metrics;
b. Technical dependencies between files;
c. Software's architectural properties;
d. Network metrics.

We extracted the technical interdependencies between files using Understand[iii], an enterprise software application that parses the lines of code saved within each file and picks up on different forms of dependencies (e.g. read, write, call functions) to other files.

The dataset also includes the following organizational variables:

a. Developers' data (gender, age, education, tenure, type of work contract, location, job title, etc.);
b. Files' authorship and ownership;
c. Tasks' characteristics, including date, time, author, and type of activities (file creation or modification);
d. Task dependencies (between pairs of files);
e. Cross-developer collaboration.

The data regarding the organizational ties among the software developers were built upon company records, elaborating on weekly data of task assignments (including task/problem description, tasks assignments to the developers, whether files were created or modified during

the performance of a given task, and the time at which the task was accomplished). Based on these records, we identified cross-developer organizational ties as occurring when two or more developers performed tasks on the same file during a given week.

These quantitative data were complemented by the information we were able to gather during our field visits, interviews, and direct observations conducted over a period of two years. More technical details about the process of data collection can be found in appendix 1.

Measures

The unit of analysis in our study is the dependency dyad between two files (design components) in the analyzed software (product design). We capture the dependencies between files by using the data of "revisions," in which project leaders together with developers identify which files are dependent on one another and should be changed together in a single task accomplishment. This approach to cross-file dependency definition is standard practice in the software engineering literature. The joint perceptions of managers and developers in judging which files depend on one another is considered an accurate estimate of the presence of actual technical dependencies between files.

Then, for each dependency dyad we measure whether it is a cross-module dependency, connecting two files belonging to separate modules. In order to identify modules, and then, cross-module dependencies, we rely on the joint perception of project leaders and developers of the software architecture, manifested in the way they cluster files and assign them into sub-directories. The structure of the software's master directory and clustering of files within sub-directories is assumed to be an accurate proxy of the software architecture, as (jointly) perceived and realized by the managers and developers. Our field visits and interviews with project leaders and developers in the studied software division confirm such assumption.

Overall, there are 94,220 unique task dependency per version in our dataset, of which 64,792 are cross-module dependencies (across sub-directories at lowest tier of the software directory). Further, we observe whether engineers collaborated while creating and/or changing (any of) the two files in the dependency dyad. This allowed to create our dependent variable.

Dependent Variable

Non-mirroring: It is a binary variable that equals 1 ("non-mirroring") when for given two dependent files (cross-module link), the file-owners did not work jointly on at least one of the files while changing them. Similarly, the dependent variable equals 0 either when the file-owners did work jointly on at least one of the two files, or when the owner of the files is the same person. A more-detailed description about how we define and measure the dependent variable ("non-mirroring" (Cataldo & Herbsleb, 2013; Cataldo, Wagstrom, Herbsleb, & Carley, 2006; Colfer & Baldwin, 2016)) can be found in appendix 3.

Independent Variables

Task innovativeness. As previously mentioned, software development tasks can be classified into two distinctive types— "creating new features," and "modifying existing features". Creating new features implies writing new lines of code possibly generating new knowledge, making sure that it fits seamlessly with the current knowledge base of the organization. Modifying existing features of the software includes local search for either incremental solutions for minor functionality issues in current features (bug fixing), or conducting slight changes in order to add minor functionalities to existing features. This type of task involves refining, recombining, or reconfiguring existing pieces of knowledge embedded in the current product design. We operationalize this as a dichotomous variable that

takes value 0 in the case of "modifying existing features," and value 1 in the case of "creating new features."

Physical proximity. This also is a dichotomous variable that captures the co-location of the team members. If the developers are located in the same office, they work close to one another, generally seated around the same table (coded as 1), whereas when engineers work remotely or from outside of the company, we consider them as not co-located with other developers (coded as 0).

Time pressure. This variable is operationalized measuring the number of weeks elapsed between the beginning of the development of a given software version and the point in time at which a focal file is created (or modified). As time passes, the deadline for the software's release approaches. Thus, the longer the elapsed time, the closer the release deadline, the higher the time pressure.

Control Variables

We control for two important organizational variables. The first is the "developers' role". This discrete variable, ranging from 0 to 6, corresponds to the team members' job titles in the software division. Higher values indicate jobs entailing more responsibilities, deeper and wider sets of programming skills as well as better knowledge of the software architecture and managerial skills. The second control variable is the developers' "type of contract": this dummy variable captures the contractual arrangement under which the developers operate. The variable takes value 0 if the developer is an employee of the company, while it takes value 1 if he/she is a contractor.

In addition to these variables, in our analyses we control for the "software lifecycle," measured as the lapse of time between the first week of developing the software (of the first

version in our sample) and the week the focal task was performed. We account for this variable to capture the fact that over the product lifecycle team members generate and share a common understanding of the product architecture, dissolving some of the uncertainties around the development process, and thus "naturally" leading to a decrease in the occurrence of "non-mirroring".

Lastly, we added a set of file-level control variables, as suggested in the software engineering literature. These variables account for differences in file content and for other peculiarities. Controlling for these variables is important in order to clearly identify the effect of the hypothesized independent variables on "non-mirroring" and distinguish them from other potential sources of variation generated by the files' heterogeneity. The first control variable, "lines of code," is a software metric used to measure the size of a file by counting the number of lines in the text of the file's source-code divided by 100, to control for its fairly large range compared to the rest of variables.

The second control variable, "hygiene of codes," measures the internal (and not structural) complexity of the file and the accessibility of the code written and saved within a given file. We used McCabe's measure of internal complexity, a standard metric in software engineering literature (Sturtevant & MacCormack, 2013). Furthermore, we account for the "comment-to-code ratio" of a focal file in order to control for possible explicit knowledge-sharing between the developers. The lines of comment contained within a focal file usually contain the guidance and/or instructions about the functionality of specific parts of the code, and mostly authored by the owner-developer of the file. It appears that, in our context, the project leaders could manage documenting and explicating the specific knowledge about internal content of files, facilitating

the transfer of knowledge to other developers for their future reference, thus reducing the need for future potential collaborations, and hence, decreasing the likelihood of non-mirroring.

Model Specification

Given the dichotomous nature of the dependent variable, non-mirroring occurrence, we use logistic models to estimate the probability of observing cases of "non-mirroring"[iv]. In addition, we used probit models as robustness checks to test the hypotheses under a slightly different assumption of the error term's distribution with steeper cumulative distribution function. These models yielded similar results as reported in the appendix 4.

Besides, to control for possible endogeneity issues, we exploited a peculiar characteristic of our data creating and estimating a diff-in-diff model. As mentioned in the previous section, after developing 4 versions of the software, the marketing department of the analyzed company requested that the software division design a radically novel version of the software, with redesigned user interfaces and many other new features. This request was fully exogenous to the software division strategy and organization, as it was forced by the competitors' launch of new and innovative versions of their software. At the same time, the software division continued to develop improvements and service packs for the old version in order to keep serving customers who chose not to upgrade. Therefore, in our setup we have two sub-datasets and use the exogenous shock corresponding to the necessity to build a new software in order to better identify our mechanisms via a diff-in-diff approach. The new branch includes data for the software versions 2.0.0 to 2.1.0 (2.X), and represents the "treated" group of files, as engendered in response to the exogenous shock. The "control" group of files includes the software versions developed in the old branch, i.e. 1.5.2 to 1.5.5 (1.5.X). As a robustness check, we also calculated the treatment effect only on the two versions developed right after the shock for each branch,

namely versions 1.5.2 and 2.0.0. The results demonstrated similar patterns found in the comparison of the two branches developed after the shock.

FINDINGS

Descriptive Statistics and Correlations

Table 2 reports the descriptive statistics of the variables. It includes the two different objects of analysis of our study, i.e. the tasks and files depending on the variables and their measures. Overall, we observe 113,610 *micro-tasks X dependency dyads* performed while developing the 13 versions of the software. Of all the observations, 67% are *cross-directory* dependencies (as above described, we use sub-directories to identify software modules). Furthermore, table 2 reports that in 22% of cases, there is no cross-developer (organizational) tie corresponding to a given cross-module technical dependency.

Moreover, approximately 60% of the development tasks include creating new files, indicating that new, innovative features are added to the software during the development process. More than 91% of the development tasks are performed by co-located developers. Furthermore, 17% of the tasks are outsourced to developers who are not company's employees. Overall, we analyze 235 weeks of development work on the studied software versions, the most challenging of which took 68 weeks of work before release (version 2.0.0).

In addition, the files contain 202 lines of code on average, with a skewed distribution, as expected. The files' hygiene of codes is impressively low, suggesting high programming skills and effective supervision by project leaders, both keeping the codes extremely clean.

---

insert table 2 about here.

Table 3 reports pair-wise correlation estimates between the main variables of the study. The sign and significance of the correlation coefficients seem to point in the direction of our hypotheses. However, the size of the correlation coefficients between the main three variables is small enough to reduce the concern of collinearity. Task innovativeness significantly correlates with the control variables, implying contingent effects on technical file characteristics based on the nature of the task performed (new feature adding versus bug fixing). Interestingly, the strong correlation between task innovativeness and the developers' type of work contract implies that tasks of adding new features are 50% more likely to be assigned to employees rather than to contract developers.

Finally, a notable observation is that time seems to have a dual effect on the software development process. Specifically, during the development of the software versions we observe a considerable increase in non-mirroring, indicative of growing complexity. This is evident in the positive and significant correlation between time pressure and non-mirroring, i.e. 0.04 with P-value smaller than 0.001. On average, in each month of a given software version development we expect to see about 16% increase in occurrence of non-mirroring. Conversely, over the course of the software lifecycle (across software versions), we see a general trend of decrease in the likelihood of occurrence of non-mirroring (estimated correlation of -0.04, with P-value <0.001). These seemingly contradictory observations motivate further examination. In the discussion section, we will revisit this finding.

Insert table 3 about here.

Hypothesis Testing

Table 4 reports the logit estimations for the first three hypotheses. The first column reports the estimations of the base model including only file-level control variables. The second column contains all control variables. The third column adds the independent variables. Our estimates support H1, H2, and H3. Task innovativeness reduces non-mirroring as well as Developers' proximity (co-location), whereas time pressure increases non-mirroring.

The fourth model adds the software versions' fixed effects in order to account for possible systematic differences between versions. In fact, this is a necessary check, because in the field interviews we conducted with the developers and managers we discovered that some of the software versions included more radical changes and new feature developments, while others were more incremental, basically entailing modifications of features inherited from previous releases. The patterns of the relationships—even after accounting for version specificities—are robust and consistent with those of the previous model, supporting hypotheses H1 and H2. Instead, we did not find significant support for hypothesis H3, most likely because of collinearity between the number of weeks passed from the beginning of developing the new version (our proxy for time pressure) and the versions' fixed effects; the versions that took longer to develop are usually those that demonstrate higher levels of non-mirroring. In the fifth model (last column of table 4) we added developers' fixed effects. Again, hypotheses H1 and H2 are supported, while H3 was not supported. Subsequent to the regression analysis, likelihood

ratio tests were run for each pair of models, whose results show continuous significant improvements in the explanatory power of subsequent models.

In addition, the results reported in models 3, 4, and 5 showed that the developers' job roles and types of contracts play a key role in explaining mismatches. These finding are consistent with extant literature, and advance our understanding of the phenomenon by providing the opportunity to compare the relative effects of the factors introduced in prior studies.

Insert table 4 about here.

Differences-in-Differences Analysis

Extant research in design literature postulates that complex designs are inert and fairly stable over time. We observe the same pattern in our dataset, where a remarkable proportion of design elements (file) hold similar design characteristics across inter-generational changes and along subsequent versions, whereas a minor proportion of files are created and/or changed. This fact might raise the issue of auto-regression in our empirical analyses, where we document similar patterns recurrently, and thus, might report biased findings. To address this issue, we leverage on an instance of radical software redesign exogenously imposed by the competitors' launch of newly designed softwares. This opportunity enables us to examine whether our findings are similar in the case where the software design is reshuffled, new design elements are created, and novel interdependencies are formed, hence the software design breaks out of its past trajectory.

As previously discussed, in order to react to competitors' innovations, the marketing department of the analyzed company requested the software division to radically redesign the software in December 2009 (week 65 in our dataset). Thanks to this shock, which we can consider exogenous to the software division, we can partition our dataset into two sub-datasets. The first dataset is for the old, existing software versions and corresponding development activities, including 1.5.X versions—which continues to undergo development as service packs and modification releases for existing, non-upgrading customers. The second sub-dataset is for the new software versions and development activities (2.X). Exploiting this event, we set up a quasi-experimental design in which the files pertaining to the new trajectory of software design evolution are impacted (or "treated") by the exogenous decision of redesign, whereas files pertaining to the releases of service packs for the earlier version function as control group. This allow us to better identify our causal relationships and the corresponding mechanisms.

Table 5 reports the results of our diff-in-diff analysis. Model 1 includes the direct "treatment" effect and demonstrates the significant systematic increase of non-mirroring in the software versions subjected to the redesign efforts. In addition, the estimation of the post-shock effect shows that after the software was redesigned more non-mirroring happened during the development of both treated and control versions, probably because of additional global complexity all over the software design. In model 2, we added the control variables to the treatment model, while model 3 delivers the full model including the treatment effect in the presence of the study's main and control variables. The results, first demonstrate that after redesigning the software architecture non-mirroring increases, probably caused by the increased complexity of the software. Moreover, it is evident that our hypotheses are supported even after the software design is reshuffled exogenously, and thus, the auto-regression issue is mitigated.

Insert table 5 about here.

Robustness Checks

As shown earlier in table 4, our findings are robust to a number of different model specifications. Models including developers' fixed-effects deliver results identical to those of the baseline model. Similarly, adding software versions' fixed effects, the results also hold, except for time pressure. As an additional robustness check, we re-ran the logistic regressions using a probit specification which assumes a slightly different distribution of error term whose cumulative distribution function is steeper than that of logit model. The findings are similar to previous results. The estimation of probit models can be found in Appendix 4.

Our findings are also robust to a different operationalization of the dependent variable. In the previous analyses, we measured non-mirroring using cross-subdirectory technical dependencies of the software master directory as a proxy for cross-module technical dependencies. This approach is widely used in the software engineering literature (also for the ease of analysis it implies). However, for the alternative operationalization of the dependent variable, we use a widely-used computational algorithm employed to outline clusters/modules in large networks: the Louvain methodology (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008). It is an iterative clustering technique that identifies clusters of nodes with the highest internal dependencies and the lowest external dependencies. It repeats this clustering procedure using an inductive approach through a series of iterative steps, therefore gradually maximizing the degree of modularity of the whole network (more details about the Louvain methodology can be found in the appendix 2).

Following this approach, we identify the software modules and, consequently, the cross-module dependencies. Then, we used the same logic described above to define the cases of non-mirroring and operationalize the dependent variable. Said differently, the dependent variable is still a binary variable that equals 1 ("non-mirroring") when for a given cross-module dependency, the file-owners did not work jointly on at least one of the files while changing them. The variable equals 0 either when the file-owners did work jointly on at least one of the two files, or when the owner of the files is the same person. But in this case modules are not defined using sub-directories but the Louvain algorithm. Our findings, reported in appendix 5, provide additional support for hypotheses 2 and 3. However, the hypothesis 1 is not supported and the results are in a somewhat opposite direction of what we hypothesized.

A reasonable way of interpreting this finding is that the process of adding new features is fully exploratory in nature, with problem solving characterized by uncertainty about the optimal solution and the most functional arrangement of corresponding interdependencies. Only through testing different solutions, developers identify cross-module links, resolve puzzles, stabilize file technical dependencies, and eventually redraw new module boundaries around newly created features. Thus, considering a more reliable measure of modularity and a more realistic representation of the degree of non-mirroring, it is reasonable to observe not only more cases of non-mirroring, but also more cases of non-mirroring when developers create new features (compared with situations in which they prevalently fix bugs or modify existing files), whose future dependencies have not emerged and/or have not been fully understood, yet.

DISCUSSION AND CONCLUSION

This study offers the first in-depth analysis of what determines non-mirroring between product and organizational architectures. It investigates why products and organizations have mismatched architectures and how they co-evolve over time. Leveraging on a unique dataset of file-level technical and organizational data on the development of an industrial software, we first identify the modules in a software architecture (using both the software directory structure and a clustering algorithm). Then, we focus on cross-module technical interdependencies and analyze the corresponding organizational dependencies among software developers, describing when they are not aligned, how such mis-alignment evolves over time, and identifying the determinants of such mis-alignment.

We apply a variety of regression techniques in order to test if and to what extent the *innovativeness* of the development tasks, as well as the *space* and *time* of task execution affect the probability of having cross-module technical dependences without corresponding organizational dependencies. We find that non-mirroring is more likely when the nature of the development task involves local search and incremental changes (modifying existing features), and less likely when the nature of the development task is more innovative (adding new features).

Moreover, we find that non-mirroring is more likely if developers are distant (not co-located) and under time pressure (the date of the software release approaches). Our findings suggest that the nature of development work and certain organizational choices might shape product architectures in the direction of higher complexity. In turn, product architectures can then shape back organizational structures as cross-modular technical dependencies (intendedly designed as cross-module interfaces or unintendedly deriving from earlier organizational

choices) increase coordination requirements and add to development work in the form of increased information sharing, communication, and collaboration.

Further, as already mentioned in the finding section, our analyses documents basic evidence of dual temporal dynamics of problem solving in knowledge-intensive firms (for reviewing the analyses pertaining to the relationship between the dependent variable, non-mirroring, and the two variables capturing dual representation of time, i.e. time pressure and software lifecycle, please refer to table 3). As time goes by during the development process of each software version, problem solving takes places in the form of addition of new features and implementing modifications of existing features. As the deadline for release approaches and time pressure mounts, though, the probability of overlooking cross-module technical dependencies increases.

However, considering the overall development process across software versions, engineers recurrent and collective problem-solving spurs collaborative learning and the formation of a shared understanding of the product architecture. Such shared knowledge improves the engineers' understanding of module boundaries and cross-module interfaces, eventually leading to a decrease in the occurrence of non-mirroring over longer term, and across generations of the product.

Thus, our findings reveal two simultaneous and conflicting dynamics through which the analyzed product architecture and corresponding organizational structure go over time: short term non-mirroring growth *during* the development of a software version and long-term uncertainty resolution *across* software generations. Our estimations provide initial evidence of such dual effect and open a potential avenue for future analysis of the co-evolution of the

product and organization architectures in contexts characterized by frequent problem solving activities and continuous knowledge generation.

Several limitations apply to this study. First, we examined organizational and technological changes in the context of a specific company, and in the software industry. While this context is replete with recurrent problem solving on a daily basis, our findings might not be generalizable to other contexts where product components are more complex, and problem solving activities are different in nature and intensity. Second, we studied a commercial software package developed in a closed enterprise setting. Therefore, our suggested organizational contingencies might not easily apply to the context of open-source software development, or similar contexts of voluntary problem solving and open organizations. Third, most of the innovative activities studied in this research were incremental rather than radical, even considering new software feature development. Our suggested implications have to be reexamined during instances of really radical and disruptive (and most likely exogenous) innovations, as the company's knowledge base renders obsolete, and therefore organizational contingencies of conducting innovation should be different.

This study opens some avenues for future research. First, we believe that using a microstructural approach to the study of modularity in products and organizations (technical dependencies at the design-element level, organizational dependencies at the task level) offers a promising perspective to understand the dynamics of technological and organizational systems complexity. Moreover, in order to replicate and validate our findings in contexts other than the software industry, future research might look into other innovation-intensive industries, reveal new contingencies, and effectively connect to performance effects in terms of quality,

time and resource efficiency and engineers' productivity, job satisfaction and capability development.

TABLES AND FIGURES

Figure 1. Types of "non-mirroring" (mismatches between task and organizational dependencies)

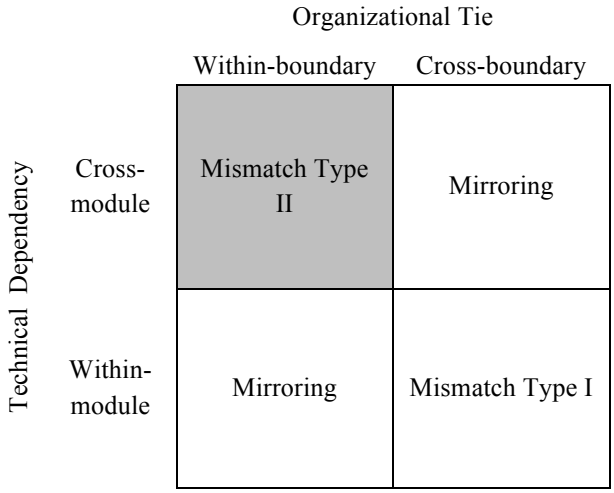


Figure 2. Progression of software versions over the period of study

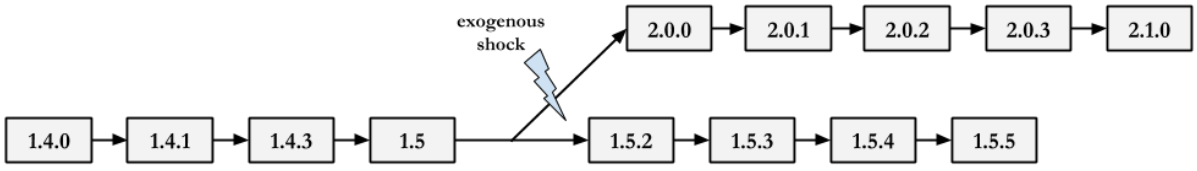


Table 1. Descriptive data about the analyzed software versions and the corresponding development process

Version	1.4	1.4.1	1.4.3	1.5	1.5.2	1.5.3	1.5.4	1.5.5	2.0.0	2.0.1	2.0.2	2.0.3	2.1.0
Release Dates	12/15/08	3/17/09	4/29/09	7/27/09	3/8/10	11/3/10	2/17/11	3/7/12	11/30/10	5/16/11	10/10/11	7/23/12	3/28/13
# of design components (files)	3152	3153	3153	3416	3603	3613	3616	3795	3868	3960	4014	4030	4149
# of technical dependencies b/w files	14887	14932	14946	18838	20032	20047	20061	20784	21067	21482	21774	22156	22914
# of tasks accomplished	1385	207	192	803	900	1346	251	917	2531	324	421	562	601
# of observations (task dependencies X files)	5732	1332	1812	29912	16328	8170	334	12444	25274	444	2152	6820	2860
# of cross-module task dependencies according to directories	4480	956	1304	16640	12018	5558	170	8874	18368	262	1072	4848	1932
# of cross-module task dependencies according to the Louvain method	3236	540	1012	9624	1784	3472	152	1372	3658	62	256	2208	986
# of developers	6	9	8	11	11	13	10	8	15	10	9	7	8

Table 2. Variables, measures, and descriptive statistics

Variable	Measure	Object of analysis	Mean	Std. Dev.	Min	Max
Dependent variable						
Non-mirroring *(Mismatch type II)*	While performing cross-directory dependent tasks, whether the owner-developers co-worked: non-mirroring = 0 if developers co-worked non-mirroring = 1 if developers did not co-work	Task	.2211501	.4150241	0	1
Independent variables						
Task Innovativeness	Task's innovative content, measured for each task: = 1, if creating new file = 0, if modifying existing file or bug-fixing	Task	.5910447	.4916432	0	1
Proximity (of Owners)	Co-location of the owners of the two files co-changed in the focal task	Task	.9138693	.2805581	0	1
Time Pressure	The lapse of time between the beginning of development process and the time when the focal task is performed (in weeks)	Task	16.49006	15.54884	0	68
Control variables						
Job Role	The job role of the developer who performed the focal task: = 0 for developer, = 1 for senior developer, = 2 for OS developer, = 3 for tester, =4 for architect, = 5 for consultant, = 6 for project leader.	Task	1.527547	1.52111	0	6
Type of Work Contract	= 0 if the developer is the company's employee, = 1 if the developer is outsourcer.	Task	.1708765	.3764027	0	1

Software Lifecycle	Number of weeks passed from the first day when developing the software began	Task	79.68963	57.02067	0	235
Lines of Code	Total number of statement lines authored in the focal file, divided by 100	File	2.023813	3.222473	0	36.72
Hygiene of codes	[Reversely coded] Internal cyclomatic complexity of the code contained in the focal file, calculated by mining the code: Low = 0, medium = 1, high = 2	File	.2783969	.5868154	0	2

Number of observations: 113,610

Table 3. Pairwise correlation results

	1	2	3	4	5	6	7	8	9	10
1. Non-mirroring (Mismatch type II)	1									
2. Task Innovativeness	-0.34	1								
3. Proximity (of owners)	-0.64	0.29	1							
4. Time Pressure	0.04	-0.03	-0.1	1						
5. Job Role	-0.11	0.2	0.12	-0.23	1					
6. Type of Work Contract	0.07	-0.48	-0.08	0.07	-0.17	1				
7. Software Lifecycle	-0.04	0.01	-0.07	0.66	-0.25	0.11	1			
8. Lines of Code	0.11	-0.27	-0.14	-0.01	-0.13	0.07	-0.02	1		
9. Hygiene of Codes [reversely coded]	0.15	-0.31	-0.15	-0.06	-0.11	0.06	-0.03	0.66	1	
10. Comment-to-code ratio	-0.08	0.04	0.07	-0.08	0.02	0.09	-0.1	-0.12	-0.11	1

All correlation coefficients are significant with 99% confidence interval or higher.

Table 4. Logit regression results for "non-mirroring"

Dependent Variable: *Non-mirroring (mismatch type II)*	(1)	(2)	(3)	(4)	(5)
Task Innovativeness			-0.527***	-0.415***	-0.458***
[adding new features = 1]			*(0.03)*	*(0.04)*	*(0.05)*
Proximity			-4.310***	-4.366***	-4.389***
[of owner-developers]			*(0.04)*	*(0.05)*	*(0.05)*
Time Pressure			0.009***	0.000	-0.001
[# of weeks passed from kick-off day]			*(0.00)*	*(0.00)*	*(0.00)*
Control variables					
Job Role		-0.202***	-0.073***	-0.174***	-0.241***
[of the developer]		*(0.01)*	*(0.01)*	*(0.01)*	*(0.01)*
Type of Work Contract		0.225***	-0.927***	-1.156***	-0.439***
[outsourcer = 1]		*(0.03)*	*(0.05)*	*(0.05)*	*(0.06)*
Software Lifecycle		-0.005***	-0.009***	-0.007***	0.002***
[# of weeks passed from the first day]		*(0.00)*	*(0.00)*	*(0.00)*	*(0.00)*
Lines of Code	0.018***	0.003	-0.013*	-0.012	0.011
[divided by 100]	*(0.00)*	*(0.00)*	*(0.01)*	*(0.01)*	*(0.01)*
Hygiene of codes	0.440***	0.378***	-0.026	-0.034	-0.220***
[reversely coded]	*(0.02)*	*(0.02)*	*(0.03)*	*(0.03)*	*(0.03)*
Comment-to-Code Ratio	-0.100***	-0.106***	-0.007*	-0.005	0.002
	(0.01)	*(0.01)*	*(0.00)*	*(0.00)*	*(0.00)*
Constant	-1.417***	-0.677***	3.344***	4.080***	4.573***
	(0.01)	*(0.03)*	*(0.06)*	*(0.08)*	*(0.12)*
Version's Fixed Effects	-	-	-	Y	-
Developer's Fixed Effects	-	-	-	-	Y
Wald Chi^2	1628.1***	2888.4***	22901.0***	24975.5***	26063.7***
Log pseudo-likelihood	-30308.538	-29678.392	-19672.092	-18634.793	-18089.766
Pseudo R^2	0.026	0.046	0.368	0.401	0.419
Number of observations	60631	60631	60631	60631	60627

*Standard errors in parentheses, * p < 0.05, ** p < 0.01, *** p < 0.001*

Table 5. Logit regression results for diff-in-diff setup

Dependent Variable: *Non-mirroring (mismatch type II)*	(1)	(2)	(3)
Treatment * post version 1.5	0.633***	0.660***	0.588***
	(0.05)	(0.05)	(0.05)
Post version 1.5	0.287***	1.115***	0.693***
	(0.03)	(0.04)	(0.05)
Innovativeness of the task			-0.714***
[adding new features = 1]			(0.03)
Proximity			-4.145***
[of owner-developers]			(0.05)
Time Pressure			0.004**
[# of weeks passed from kick-off day]			(0.00)
Job Role		-0.193***	-0.066***
[of the developer]		(0.01)	(0.01)
Type of Work Contract		-0.000	-1.158***
[outsourcer = 1]		(0.03)	(0.05)
Software Lifecycle		-0.010***	-0.012***
[# of weeks passed from the first day]		(0.00)	(0.00)
Lines of Code		-0.013***	-0.029***
[divided by 100]		(0.00)	(0.01)
Hygiene of codes		0.466***	0.125***
[*reversely coded]*		(0.02)	(0.03)
Comment-to-Code Ratio		-0.206***	-0.026***
		(0.02)	(0.00)
Constant	-1.869***	-1.301***	2.890***
	(0.03)	(0.03)	(0.07)
Wald Chi^2	1708.350***	5597.451***	10817.698***
Log pseudo-likelihood	-34850.844	-32559.691	-22788.046
Pseudo R^2	0.024	0.088	0.362
Number of observations	71840	71840	71840

*Standard errors in parentheses, * p < 0.05, ** p < 0.01, *** p < 0.001*

APPENDICES

Appendix 1. Data Collection Process

The process of data collection began by signing a non-disclosure agreement with the company of our study and obtaining exclusive access to the repository of working copies of each version of the software and the data of its development, task assignment, and task accomplishment.

We analyzed the source codes and files of each version using the Understand software package in order to extract the dependencies between files, allowing us to construct the network of dependencies between software components, which is the architecture of the analyzed software in a given version. It is noteworthy that not all kinds of the technical dependencies found in the code affect the functioning of a given dependent file, and therefore, in our dataset we disregard them as a technical interdependency. For example, code dependencies such as "import" in Python language or "include" in C language, which only reports which global libraries are used in the focal file, are not accounted as a dependency in our dataset, because such dependencies do not constrain the functionality, and so, design decisions in the dependent file.

The edge list of dependencies between files was analyzed by Hidden Structure methodology (Baldwin et al., 2014) to calculate internal and structural characteristics of the files. Further, as an alternative operationalization of dependent variable, we partitioned files into modules displaying highest internal cohesion and lowest external dependencies. To this purpose we processed the edge list of files' dependencies using the "Louvain methodology" to attribute each file to a single module, or using graph analysis terminology, "partition". The analyses were run in Python environment and the outcome was saved back to the main dataset. The Louvain methodology is a general computational algorithm in graph analyses allowing for

community/cluster/module detection in large networks. It is proved to be the fastest and most accurate computational method to identify modules in networks consisting of thousands of nodes and hundreds of thousands of ties, which is a characteristic of our dataset as well (more information about the Louvain methodology can be found in Appendix 2). Based on module membership data, we calculated the number of cross-module dependencies for a given file, upon which we created the alternative dependent variable measuring non-mirroring based on new identification of cross-module technical dependencies.

In the next step, we added revision data to our database. It includes task-specific data of author, date, time, etc. Each single "revision," representing a macro-task assignment and accomplishment in this context, includes a set of tasks on the files that were called, changed, and returned to the software repository by a single developer. Therefore, our dataset articulates each "revision" into tasks for the files called within a given "revision."

The data of task dependencies, then, are constructed using the following steps:

1. Extracting the data of tasks performed in each "revision" from the revision data repository of the company.
2. Creating the bipartite matrix of files that were changed in each revision
3. Calculating the one-mode matrix of the files that were changed together in the same "revision", using Python's NetworkX library.
4. Merging back the data of task dependencies into the main dataset.

Appendix 2. An overview of Louvain method of identifying modules in complex networks

Any technical dependency is comprised of two dependent files. Since this research focuses on non-mirroring as defined for cross-module dependencies, it is essential to identify whether two dependent files belong to the same module, or lie in two different modules. A module includes design components that render maximum internal coherence and minimum external connectivity (Alexander, 1964; Parnas, 1972). It has been theoretically argued and empirically demonstrated that the nature of dependencies and their technical and organizational properties are different from each other. Consistent with this stream of research, we have accounted for these architectural characteristics of dependencies in our study.

In order to identify the modules in the analyzed product (software) architecture, we benefited from recent mathematical developments in graph theory and novel improvements in calculation methods developed in the network analysis research. To outline clusters within graphs formally represented in the form of dependency matrices, graph theory suggests graph spectra analyses in order to partition the matrices (Lee, Hoehn-Weiss, & Karim, 2016). However, conducting these analytical algorithms that are mostly based on eigenstructure analyses is computationally onerous and time-consuming when trying to conduct the analysis on very large graphs and networks. Therefore, network scientists have developed more parsimonious computational techniques in order to address this difficulty. Among these techniques, we opted for the Louvain algorithm (Blondel et al., 2008) in order to identify modules (clusters) within a large network of components and dependencies. The Louvain algorithm is based on modularity optimization. It is shown that this technique outperforms other currently-known community-detection methods in terms of computational accuracy and time, when used for very large networks of nodes that are dependent on one another (Blondel et al., 2008).

The method consists of two phases. First, it looks for small modules (also referred to as clusters or communities) by optimizing modularity in a local way. Then it aggregates nodes of the same community and builds a new network whose nodes are the communities. These steps are repeated iteratively until a maximum of modularity is attained. The modularity of a partition is a scalar value between -1 and 1 that measures the density of dependencies inside communities as compared to dependencies between communities.

$$Q = \frac{1}{2m}\sum_{i,j}\left[A_{ij} - \frac{K_i K_j}{2m}\right]\delta(c_i, c_j),$$

where A_{ij} represents the weight of the link between *i* and *j*, $k_i = \sum_j A_{ij}$ is the sum of the weights of the edges attached to vertex *i*, c_i is the community to which vertex *i* is assigned, the δ-function δ(u, v) is 1 if u = v and 0 otherwise, and $m = \frac{1}{2}\sum_{ij} A_{ij}$ (Blondel et al., 2008).

Appendix 3. Measure of "non-mirroring"

As summarized in Figure 1, "mirroring" takes place when the organizational ties in the form of collaboration among developers overlap with the relevant dependencies between design components. This allows for efficiently allocating the developers' scarce cognitive resources to the necessary tasks (within or between modules) and rightsize the amount of cross-module information sharing and information processing according to the extent to which it is confined in standardized interfaces. Developers who perform tasks on design components (files) belonging to a specific product module, can focus on components within the module and disregard other components with obvious benefits in terms of design accuracy and efficiency. Developers who perform tasks on design components (files) belonging to separate modules, should span the module knowledge and technical boundaries, interacting with other developers

and/or teams in order to coordinate the design decisions, communicate the information required for performing the tasks at hand, and accordingly collaborate on carrying out the cross-module task dependencies.

However, developers might interact even if this is not necessary to perform their tasks. For example, when they talk to developers of design components (files) pertaining to another module in the absence of cross-module technical dependencies, this would represent a redundant and inefficient tie (mismatch type I), since it unnecessarily involves two developers (or teams) while the task could be effectively performed without recourse to interaction.

Similarly, if the developers do not interact when it *is* necessary, i.e. when they do not talk to developers of design components (files) pertaining to another module in the presence of cross-module technical dependencies, this would represent a problem, potentially generating future technical debt and rework, with the organization incurring significant costs due to the lack of coordination (mismatch type II). Without the necessary collaboration, the sole developer would either should independently obtain required knowledge for changing the component that spans the boundary of his/her own modules, or, even more problematically, simply ignore the cross-module technical dependency. Such lack of interaction, information sharing, and collaboration could eventually harm the product's functionality.

Following this logic, we focus on the correspondence between the technical dependencies and the organizational ties among developers (Cataldo & Herbsleb, 2013; Cataldo et al., 2006; Colfer & Baldwin, 2016). In our study, technical dependencies are inferred from the task dependencies. This is a common approach in the software engineering literature. In order to identify modules, and so, cross-module dependencies, we rely on the joint perception of the project leaders and developers in setting the boundaries around files in the form of directories.

More specifically, we assume that the structure of the analyzed software's development directory is a reasonable proxy of the software architecture, focusing on the cross-directory task dependencies (as proxies of cross-module technical dependencies), and examine whether the two corresponding developers collaborated while working on the given task.

We construct task dependencies based on the joint-partitioning of the tasks by project managers and developers. In the context of software development, each task—usually meant to either add a new feature/function or improve an existing solution—includes adding or changing a few interdependent files at once in a single task. Such task, in the software development jargon, is often referred to as a "revision." We use the detailed data available on revisions in our dataset in order to capture task dependencies between design components (i.e. files). As previously mentioned, any revision entails changes to a set of dependent files. These dependencies either relate to files that belong to the same module or encompass files belonging to two or more modules. We use this data to detect within-module and cross-module dependencies.

Alternatively, we detect organizational ties, i.e. collaborations between developers, when two developers work simultaneously on the same file. In our research setting it is common practice that the appropriate window of time to capture cross-developer collaboration is a week, since development work is planned and tasks are assigned on a weekly basis by project managers. Therefore, for each given file we record whether or not there has been a collaboration between developers within a given week.

Eventually, for each pair of dependent components we: a) examine whether the tasks accomplished are within- or cross-module; and b) record whether the pair of dependent design components (files) are changed by one or more developers during the same time window. Combining these two dimensions (technical and organizational dependencies) we obtain a 2x2

matrix which captures if an organizational dependency corresponds to a given technical dependency (quadrants 1 and 3 in figure 1), or if the two sets of dependencies do not mirror one another. This could be due to a lack of cross-developer collaboration corresponding to the cross-module dependency (quadrant 2: mismatch type II, the focus of this study), or because the two developers (unnecessarily) collaborate on a given task that is self-contained within a single module (quadrant 4: mismatch type I, which is beyond the scope of this study, and hence, we disregard it).

Consequently, "non-mirroring," our dependent variable, is a binary variable that equals 1 when for given two dependent files (cross-module link), the file-owners did not work jointly on at least one of the files while changing them. Similarly, the dependent variable equals 0 either when the file-owners did work jointly on at least one of the two files, or when the owner of the files is the same person.

The unit of analysis is the dyad (pair) of dependent files. From all possible dependencies between files we first subsample dyads that correspond to task assignments and accomplishments, i.e. revisions. As previously stated, each task accomplishment (revision) in the context of software development usually involves changing a few dependent files at once. Therefore, in our new subsample we only admit files that have been changed. Then, based on the co-changes of the files in a given task we identify their task dependencies. The files that were changed in a single task are dependent on one another as perceived by the project manager who assigned the task, and/or by the developer who calls and changes them in a single task. After forming the new subset of file dyads, we measure whether the two files belong to the same module (directory) or to two separate directories.

Then, in order to measure the collaboration between the developers on a task dependency, we record whether either of the two files in a given task dependency is changed by multiple

developers at the same time. In software development, as suggested by the software engineering literature, joint-working on a given file typically implies active communication between the developers. This assumption was also confirmed during the interviews we conducted with the developers and the chief software engineer.

After capturing the collaboration data per each task dependency dyad and merging this dataset back to the data generated in the last step, we obtained our final full dataset. Then, referring to the figure 1, we identified whether a given organizational dependency (developers that collaborate) corresponds to a technical dependency (mirroring) or not (non-mirroring).

As previously highlighted, in this study we consider only the mismatch type II and refer to it as non-mirroring in our analyses. Performing tasks on a design component whose dependencies cut across the module boundaries is a demanding endeavor, requiring a sufficient acquaintance with the dependent components, their functionalities, and their interdependencies to other elements of the software. Therefore, in order to coordinate the cross-module tasks, developers or teams have to actively communicate with one another, share information, and transfer the necessary knowledge. In other words, developers and teams performing tasks that span modules' boundaries need to interact and collaborate (form organizational ties). Failing to establish organizational ties (in either form of communication, collaboration, physical collocation, or co-membership to the same project, team, or organization), as discussed before, usually leads to problems in the development process (delays, rework, etc.), and, eventually potentially defective products (software). In summary, in our study non-mirroring consists of the lack of such organizational dependency.

Appendix 4. Probit regression results for "non-mirroring"

Dependent Variable: *Non-mirroring (mismatch type II)*	(1)	(2)	(3)	(4)	(5)
Task Innovativeness			-0.378***	-0.333***	-0.243***
[adding new features = 1]			(0.02)	(0.02)	(0.02)
Proximity			-2.397***	-2.409***	-2.308***
[of owner-developers]			(0.02)	(0.02)	(0.02)
Time Pressure			0.007***	0.001	0.001
[# of weeks passed from kick-off day]			(0.00)	(0.00)	(0.00)
Job Role		-0.065***	-0.002	-0.055***	-0.062***
[of the developer]		(0.00)	(0.00)	(0.00)	(0.00)
Type of Work Contract		0.257***	-0.451***	-0.597***	-0.218***
[outsourcer = 1]		(0.02)	(0.02)	(0.02)	(0.03)
Software Lifecycle		-0.002***	-0.004***	-0.004***	-0.000
[# of weeks passed from the first day]		(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code	0.004	-0.003	-0.015***	-0.017***	-0.001
[divided by 100]	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Hygiene of codes	0.280***	0.261***	0.044**	0.068***	-0.048**
[reversely coded]	(0.01)	(0.01)	(0.01)	(0.02)	(0.02)
Comment-to-Code Ratio	-0.054***	-0.061***	-0.008***	-0.009***	-0.000
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Constant	-0.901***	-0.675***	1.670***	2.132***	1.936***
	(0.01)	(0.01)	(0.03)	(0.04)	(0.05)
Version's Fixed Effects	-	-	-	Y	-
Developer's Fixed Effects	-	-	-	-	Y
Wald Chi^2	1921.906***	2727.683***	24762.167***	27771.460***	27818.599***
Log pseudo-likelihood	-34755.229	-34352.341	-23335.099	-21830.452	-21805.121
Pseudo R^2	0.027	0.038	0.347	0.389	0.389
Number of observations	71840	71840	71840	71840	71832

*Standard errors in parentheses, * $p < 0.05$, ** $p < 0.01$, *** $p < 0.001$*

Appendix 5. Rare logistic regression results for the alternative operationalization of dependent variable based on Louvain method of module identification

Dependent Variable: *Non-mirroring (mismatch type II)*	(1)	(2)	(3)	(4)
Task Innovativeness		0.646***	0.860***	0.659***
[adding new features = 1]		(0.03)	(0.04)	(0.04)
Proximity		-0.089**	-0.157***	-0.208***
[of owner-developers]		(0.03)	(0.04)	(0.04)
Time Pressure		0.008***	0.014***	0.011***
[# of weeks passed from kick-off day]		(0.00)	(0.00)	(0.00)
Job Role	-0.137***	-0.117***	-0.135***	-0.143***
[of the developer]	(0.01)	(0.01)	(0.01)	(0.01)
Type of Work Contract	-0.203***	-0.073	-0.002	-0.127**
[outsourcer = 1]	(0.04)	(0.04)	(0.04)	(0.05)
Software Lifecycle	0.001***	-0.000	-0.007***	0.000
[# of weeks passed from the first day]	(0.00)	(0.00)	(0.00)	(0.00)
Owner-developer	0.515***	0.423***	0.447***	0.337***
[owner and developer of the focal file are the same = 1]	(0.03)	(0.03)	(0.04)	(0.04)
Lines of Code	0.031***	0.032***	0.022***	0.006
[divided by 100]	(0.01)	(0.01)	(0.01)	(0.01)
Hygiene of codes	0.262***	0.209***	0.226***	0.136***
[reversely coded]	(0.03)	(0.03)	(0.03)	(0.03)
Comment-to-Code Ratio	-0.283***	-0.182***	-0.182***	-0.171***
	(0.02)	(0.02)	(0.02)	(0.02)
Intercept	1.258***	1.057***	2.385***	1.278***
	(0.04)	(0.04)	(0.10)	(0.16)
Versions' Fixed Effects	-	-	Y	-
Developers' Fixed Effects	-	-	-	Y
Wald Chi^2	1244.913***	1744.504***	2924.777***	2660.136***
Log pseudo-likelihood	-6333.842	-6084.046	-5452.597	-5534.508
Pseudo R^2	0.089	0.125	0.211	0.194
Number of observations	25625.000	25625.000	25115.000	24507.000

*Standard errors in parentheses, * p < 0.05, ** p < 0.01, *** p < 0.001*

REFERENCES

Alexander, C. 1964. Notes of the Synthesis of Form. ***Vasa***.

Austin, R. D. 2001. The Effects of Time Pressure on Quality in Software Development: An Agency Model. ***Information Systems Research***, 12(2): 195–207

Baldwin, C., & Clark, K. 1997. Managing in an age of modularity. ***Harvard Business Review***, (October 1997).

Baldwin, C., MacCormack, A., & Rusnak, J. 2014. Hidden structure: Using network methods to map system architecture. ***Research Policy***, 43(8): 1381–1397

Baldwin, C. Y., & Clark, K. B. 2000. ***Design rules: The power of modularity***, vol. 1. The MIT Press

Baumann, O., & Siggelkow, N. 2013. Dealing with Complexity: Integrated vs. Chunky Search Processes. ***Organization Science***, 24(1): 116–132

Betz, S., Fricker, S., Moss, A., Afzal, W., Svahnberg, M., et al. 2013. An Evolutionary Perspective on Socio-Technical Congruence: The Rubber Band Effect. ***Replication in Empirical Software Engineering Research (RESER), 2013 3rd International Workshop on***, 15–24. IEEE

Blondel, V. D., Guillaume, J.L., Lambiotte, R., & Lefebvre, E. 2008. Fast unfolding of communities in large networks. ***Journal of Statistical Mechanics: Theory and Experiment***, 2008(10): 6

Browning, T. R. 2001. Applying the design structure matrix to system decomposition and integration problems: a review and new directions. ***IEEE Transactions on Engineering Management***, 48(3): 292–306

Brusoni, S., & Prencipe, A. 2001. Managing Knowledge in Loosely Coupled Networks: Exploring The Links Between Product and Knowledge Dynamics. ***Journal of Management Studies***, 38(7): 1019–1035

Brusoni, S., & Prencipe, A. 2001. Unpacking the black box of modularity: technologies, products and organizations. ***Industrial and Corporate Change***, 10(1): 179–205

Brusoni, S., & Prencipe, A. 2011. Patterns of modularization: The dynamics of product architecture in complex systems. ***European Management Review***, 8: 67–80

Brusoni, S., Prencipe, A., & Pavitt, K. 2001. Knowledge Specialization, Organizational Coupling, and the Boundaries of the Firm: Why Do Firms Know More Than They Make? ***Administrative Science Quarterly***, 46(4): 597–621

Cabigiosu, A., & Camuffo, A. 2012. Beyond the "Mirroring" Hypothesis: Product Modularity and Interorganizational Relations in the Air Conditioning Industry. ***Organization Science***, 23(3): 686–703

Cabigiosu, A., & Camuffo, A. 2016. Measuring Modularity: Engineering and Management

Effects of Different Approaches. ***IEEE Transactions on Engineering Management***, PP(99): 1–12

Cabigiosu, A., Camuffo, A., Cabigiosu, A., & Camuffo, A. 2012. ***Beyond the " Mirroring " Hypothesis : Product Modularity and Interorganizational Relations in the Air Conditioning Industry***, (February 2015)

Cataldo, M., Herbsleb, J., & Carley, K. 2008. Socio-technical congruence: a framework for assessing the impact of technical and work dependencies on software development productivity. ***Symposium on Empirical Software,*** (March): 1–20

Cataldo, M., & Herbsleb, J. D. 2013. Coordination breakdowns and their impact on development productivity and software failures. ***IEEE Transactions on Software Engineering***, 39(3): 343–360

Cataldo, M., Wagstrom, P. A., Herbsleb, J. D., & Carley, K. M. 2006. Identification of coordination requirements: Implications for the Design of collaboration and awareness tools. ***Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW***

Christensen-Szalanski, J. J. 1980. A further examination of the selection of problem-solving strategies: The effects of deadlines and analytic aptitudes. ***Organizational Behavior and Human Performance***, 25(1): 107–122

Colfer, L., & Baldwin, C. 2016. The Mirroring Hypothesis: Theory, Evidence and Exceptions. ***Industrial and Corporate Change***, 25(5): 709–738

Conway, M. E. 1968. How do committees invent? ***Datamation***

Cyert, R. M., & March, J. G. 1963. ***A behavioural theory of the firm***

Dybå, T., & Dingsøyr, T. 2008. Empirical studies of agile software development: A systematic review. ***Information and Software Technology***, 50(9–10): 833–859

Eppinger, S. D., & Browning, T. R. 2012. ***Design structure matrix methods and applications***. MIT press

Ethiraj, S. K., & Levinthal, D. 2004. Bounded rationality and the search for organizational architecture: An evolutionary perspective on the design of organizations and their evolvability. ***Administrative Science Quarterly***, 49(3): 404–437

Fine, C. H. 1998. ***Clockspeed: Winning industry control in the age of temporary advantage***. Basic Books

Fixson, S. K., & Park, J.-K. 2008. The power of integrality: Linkages between product architecture, innovation, and industry structure. ***Research Policy***, 37(8): 1296–1316

Furlan, A., Cabigiosu, A., & Camuffo, A. 2014. When the mirror gets misted up: Modularity and technological change. ***Strategic Management Journal***, 35(6): 789–807

Garud, R., & Kumaraswamy, A. 1993. Changing competitive dynamics in network industries: An exploration of Sun Microsystems' open systems strategy. ***Strategic Management***

Journal, 14(5): 351–369

Gokpinar, B., Hopp, W. J., & Iravani, S. M. R. 2010. The impact of misalignment of organizational structure and product architecture on quality in complex product development. ***Management Science***, 56(3): 468–484

Guo, F., & Gershenson, J. K. 2003. Comparison of modular measurement methods based on consistency analysis and sensitivity analysis. ***ASME 2003 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference***, 393–401. American Society of Mechanical Engineers

Hao, B., Feng, Y., & Frigant, V. 2015. Rethinking the "mirroring" hypothesis: implications for technological modularity, tacit coordination, and radical innovation. ***R&D Management***, n/a-n/a

Heaton, A. W., & Kruglanski, A. W. 1991. Person perception by introverts and extraverts under time pressure: Effects of need for closure. ***Personality and Social Psychology Bulletin***, 17(2): 161–165

Henderson, R., & Clark, K. 1990. Architectural innovation: the reconfiguration of existing product technologies and the failure of established firms. ***Administrative Science Quarterly***, 35(1): 9–30

Herbsleb, J. D. 2007. Global software engineering: The future of socio-technical coordination. ***2007 Future of Software Engineering***, 188–198. IEEE Computer Society

Hoegl, M., & Proserpio, L. 2004. Team member proximity and teamwork in innovative projects. ***Research Policy***, 33(8): 1153–1165

Hoetker, G. 2006. Do modular products lead to modular organizations? ***Strategic Management Journal***, 27(6): 501–518

Hsuan, J. 1999. Impacts of supplier–buyer relationships on modularization in new product development. ***European Journal of Purchasing & Supply Management***, 5(3): 197–209

Huang, C.-C., & Kusiak, A. 1998. Modularity in design of products and systems. ***Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on***, 28(1): 66–77

Karim, S., & Kaul, A. 2015. Structural Recombination and Innovation: Unlocking Intraorganizational Knowledge Synergy Through Structural Change. ***Organization Science***, 26(2): 439–455

Kraut, R. E., & Streeter, L. A. 1995. Coordination in software development. ***Communications of the ACM***, 38(3): 69–82

Kruglanski, A. W., & Freund, T. 1983. The freezing and unfreezing of lay-inferences: Effects on impressional primacy, ethnic stereotyping, and numerical anchoring. ***Journal of Experimental Social Psychology***, 19(5): 448–468

Kwan, I., Cataldo, M., & Damian, D. 2012. Conway's Law Revisited: The Evidence For a Task-based Perspective. ***IEEE Software***, 29(1): 1–4

Larses, O., & Blackenfelt, M. 2003. Relational reasoning supported by quantitative methods for product modularization. ***Proceedings of ICED 03, the 14th International Conference on Engineering Design, Stockholm***

Lee, C.-H., Hoehn-Weiss, M. N., & Karim, S. 2016. Grouping interdependent tasks: Using spectral graph partitioning to study complex systems. ***Strategic Management Journal***, 37(1): 177–191

Levinthal, D. A., & March, J. G. 1993. The myopia of learning. ***Strategic Management Journal***, 14: 95–112

Levinthal, D., & March, J. G. 1981. A model of adaptive organizational search. ***Journal of Economic Behavior & Organization***, 2(4): 307–333

Levitt, B., & March, J. G. 1988. Organizational learning. ***Annual Review of Sociology***, 14: 319–340

MacCormack, A., Baldwin, C., & Rusnak, J. 2012. Exploring the duality between product and organizational architectures: A test of the "mirroring" hypothesis. ***Research Policy***, 41(8): 1309–1324

MacCormack, A., Rusnak, J., & Baldwin, C. Y. 2006. Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code. ***Management Science***, 52(7): 1015–1030

MacDuffie, J. P. 2013. Modularity-as-Property, Modularization-as-Process, and "Modularity"-as-Frame: Lessons from Product Architecture Initiatives in the Global Automotive Industry. ***Global Strategy Journal***, 3(1): 8–40

Martin, M. V, & Ishii, K. 2002. Design for variety: developing standardized and modularized product platform architectures. ***Research in Engineering Design***, 13(4): 213–235

Mikkola, J. H. 2006. Capturing the Degree of Modularity Embedded in Product Architectures. ***Journal of Product Innovation Management***, 23(2): 128–146

Momme, J., Moeller, M. M., & Hvolby, H.-H. 2000. Linking modular product architecture to the strategic sourcing process: case studies of two Danish industrial enterprises. ***International Journal of Logistics***, 3(2): 127–146

Moore, D. A., & Tenney, E. R. 2012. Time pressure, performance, and productivity. ***Research on Managing Groups and Teams***, vol. 15: 305–326

Murmann, J. P., & Frenken, K. 2006. Toward a systematic framework for research on dominant designs, technological innovations, and industrial change. ***Research Policy***, 35(7): 925–952

Ocasio, W. 1997. Towards an attention-based view of the firm. ***Strategic Management Journal***, 187–206

Ocasio, W. 2010. Attention to Attention. ***Organization Science***, 22(5): 1286–1296

Orton, J. D., & Weick, K. E. 1990. Loosely coupled systems: A reconceptualization. ***Academy***

of Management Review.

Parnas, D. L. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. ***Communications of the ACM***, 15(12): 1053–1058

Pikkarainen, M., Haikara, J., Salo, O., Abrahamsson, P., & Still, J. 2008. The impact of agile practices on communication in software development. ***Empirical Software Engineering***, 13(3): 303–337

Pimmler, T. U., & Eppinger, S. D. 1994. ***Integration analysis of product decompositions***

Rivkin, J. W., & Siggelkow, N. 2003. Balancing Search and Stability: Interdependencies Among Elements of Organizational Design. ***Management Science***, 49(3): 290–311

Salvador, F. 2007. Toward a product system modularity construct: literature review and reconceptualization. ***Engineering Management, IEEE Transactions on***, 54(2): 219–240

Sanchez, R., & Mahoney, J. T. 1996. Modularity, flexibility and knowledge management in product and organization design. ***Strategic Management Journal***, 17(SI): 63–76

Schilling, M. 2000. Toward a general modular systems theory and its application to interfirm product modularity. ***Academy of Management Review***, 25(2): 312–334

Schilling, M. A. 2003. Technological leapfrogging: Lessons from the US videogame industry. ***California Management Review***, 45(3): 6–32

Siggelkow, N., & Levinthal, D. A. 2003. Temporarily divide to conquer: Centralized, decentralized, and reintegrated organizational approaches to exploration and adaptation. ***Organization Science***, 14(6): 650–669

Simon, H. 1962. The architecture of complexity. ***Proceedings of the American Philosophical Society***, 106(6): 467–482

Simon, H. A. 1972. Theories of bounded rationality. ***Decision and Organization***, 1(1): 161–176

Sosa, M. E., Eppinger, S. D., & Rowles, C. M. 2003. Identifying modular and integrative systems and their impact on design team interactions. ***Journal of Mechanical Design, Transactions of the ASME***

Sosa, M. E., Eppinger, S. D., & Rowles, C. M. 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. ***Management Science***, 50(12): 1674–1689

Srikanth, K., & Puranam, P. 2011. Integrating distributed work: Comparing task design, communication, and tacit coordination mechanisms. ***Strategic Management Journal***, 32(February): 849–875

Srikanth, K., & Puranam, P. 2014. The Firm as a Coordination System: Evidence from Software Services Offshoring. ***Organization Science***, (February).

Staudenmayer, N., Tripsas, M., & Tucci, C. L. 2005. Interfirm Modularity and Its Implications

for Product Development. ***Journal of Product Innovation Management***, 22(4): 303–321

Sturtevant, D., & MacCormack, A. 2013. The Impact of System Design on Developer Productivity. ***Academy of Management Conference. Orlando (USA)***

Sturtevant, D., MacCormack, A., Magee, C., & Baldwin, C. 2013. ***Technical Debt in Large Systems : Understanding the cost of software complexity***. MIT

Suárez, F., & Utterback, J. 1995. Dominant designs and the survival of firms. ***Strategic Management Journal***, 16(6): 415–430

Ulrich, K. 1995. The role of product architecture in the manufacturing firm. ***Research Policy***

Valetto, G., Helander, M., Ehrlich, K., Chulani, S., Wegman, M., et al. 2007. Using software repositories to investigate socio-technical congruence in development projects. ***Proceedings of the Fourth International Workshop on Mining Software Repositories***, 25. IEEE Computer Society

von Hippel, E. 1990. Task partitioning: An innovation process variable. ***Research Policy***, 19(5): 407–418

Whitfield, R. I., Smith, J. S., & Duffy, A. B. 2002. Identifying Component Modules. In J. S. Gero (Ed.), ***Artificial Intelligence in Design '02***: 571–592. Dordrecht: Springer Netherlands.

ESSAY 3

Organizational Search Strategy: An Examination of Interdependencies, Locus and Temporality of Search

ABSTRACT

Notwithstanding the ample research on organizational search and its performance implications, the factors that shape the organizations' search strategy is fairly under-explored. This study investigates how *problems' characteristics* influence the managerial decision of searching jointly or independently for solving a given problem. We make a case where problem-solvers' activities affect the landscape of the organization's future search. The empirical findings from developing an industrial software package across 13 versions and along 60 months demonstrate that the joint search is pursued when solving complex problems, involving cross-module interdependencies, whereas the problem-solvers opt for independent search efforts while searching in distant loci, discovering novel solutions. Moreover, the results show that in the short term (*within* stages of search) the software developers go for joint search, whereas in the long term (*across* stages of search) they resort to joint search activities. These insights complement the extant innovation and search literature by endogenizing the organizational search strategy decision, and thereby, exploring its determinants in a knowledge-intensive context.

Keywords: Organizational search; search strategy; joint search; complexity; interdependencies; software industry

INTRODUCTION

Innovation is the process of defining new problems and searching for novel solutions, mostly by generating new knowledge (Nonaka, 1994). Whereas the innovation literature extensively recognizes the significance of search, yet the question that how organizations should manage their search processes remains relatively unexplored (Baumann & Siggelkow, 2013, p. 116). Not until recently, studies begin highlighting the previously overlooked importance of coordinated (or joint)

search and coupled learning, despite its relevance to the theory of organizational search that naturally involves multiple actors (Knudsen & Srikanth, 2014; Puranam & Swamy, 2016). While such studies inform the performance implications, outcomes, and contingencies of joint search, the factors that shape such strategic decisions is still to be investigated.

Exploring the antecedents of organizations' search strategy (independent vs. joint search) have immediate practical significance and managerial implication. Recent observations from the modern firms highlight a critical problem of "collaborative overload" (Cross, Rebele, & Grant, 2016). Firms, following the trend to revolutionize the workplace towards promoting more collaborations of employees have reduced physical barriers, provided open spaces, and encouraged ad-hoc communications. However, it appears that firms have over-done it and the employees are overloaded by the inefficient collaborations, requiring unnecessary meetings and coordination efforts, and redundant communications. This leaves the employees with less time to perform the assigned tasks, leading to more stress, and hence, reducing their productivity, and eventually, giving rise to their burn out. The nowadays reality of workplaces is well documented in a recent Harvard Business Review article:

> *"Collaboration is taking over the workplace. As business becomes increasingly global and cross-functional, silos are breaking down, connectivity is increasing, and teamwork is seen as a key to organizational success. […] How much time do people spend in meetings, on the phone, and responding to e-mails? At many companies the proportion hovers around 80%, leaving employees little time for all the critical work they must complete on their own. Performance suffers as they are buried under an avalanche of requests for input or advice, access to resources, or attendance at a meeting. They take assignments home, and soon, according to a large body of evidence on stress, burnout and turnover become real risks." (Cross et al., 2016, p. 57)*

Managing collaboration overload is specifically more pressing in the knowledge-intensive firms. Knowledge workers, through constant search and daily problem-solving activities, develop new bodies of knowledge and incorporate them into the firm's existing knowledge base. Therefore, the firm's knowledge base, which forms the firm's search landscape for further search endeavors, is ever-changing as endogenously shaped by the knowledge workers' activities. Despite the relevance of this phenomenon, the research to inform manager's decision of search strategy in such complicated situations is underdeveloped. Most of the studies examining search processes assume the search space to be exogenously given, remaining unchanged during at least one stage of search expedition (Gavetti, Helfat, & Marengo, 2016).

An important implication of examining the endogenous evolution of problem space while studying the antecedents of firms' search strategy is the dual dynamic nature of search efforts. The constant problem-solving activities in knowledge-intensive contexts involve two conflicting temporal dynamics, which makes it unclear that which search strategy is optimal under what condition.

First, the continuous search and exploration of new domains and annexing them to the firm's existing search landscape increase the span and complexity of the firm's search space over time. Hence, knowledge workers need to constantly update their understanding of newly formed interdependencies and architecture of knowledge base, as it will bind their future search efforts. Thus, in order to efficiently perform their problem-solving activities, and seamlessly embed the pieces of knowledge they generate for solving a problem, the knowledge workers need to relentlessly update their understanding of the ever-changing architecture of knowledge and its interdependencies as shaped by other workers' activities. The knowledge workers, then, resort to

more collaborative search, in the hope of keeping their architectural knowledge up-to-date. Thus, over time, the need for collaboration will increase.

On the contrary, over repeated search and problem-solving activities, knowledge workers develop a shared understanding of the existing (and unchanged) firm's problem space, leading to uncertainty resolution about underlying interdependencies. This, in turn, provides the opportunity for tacit coordination of tasks, reducing the explicit coordination requirements and collaboration needs over time. Moreover, continuous cooperation with team members over repeated activities resolves uncertainties about the cooperative behavior of co-workers, diminishing the space for unidentifiable opportunistic behavior, and eventually, reducing the epistemic interdependence between knowledge workers. Thus, over time, the need for collaboration will decrease.

To address the mentioned shortcomings in the organizational search literature, we examine the antecedents of firms' search strategy in situations where the search space is constantly evolving and endogenously shaped by the search efforts of firms' knowledge workers. Specifically, we shed light on the phenomenon, by introducing the dual effects of time progression on search strategy, one increasing collaboration requirements *within* stages of search and the other decreasing collaboration requirements *across* stages of the search.

To do so, we leverage on our detailed micro-level dataset of daily problem-solving activities of a team of software engineers, developing an industrial software package over 60 months. For each act of problem-solving performed on a design element (i.e. files contained in the software system), we document the problem's interdependencies (i.e. total number of cross-module dependencies), locus of search (i.e. distant search by creating new solutions vs. local search by

enhancing existing solutions), lapse of the time (both within a given stage of search and across search stages), and eventually, search strategy (i.e. independent vs. joint search).

Our findings, robust to alternative model specifications, contribute to the search literature by endogenizing the evolution of problem landscape, as the interdependencies are formed by knowledge-workers' decisions and problem-solving activities. Moreover, we contribute to innovation management literature by examining the under-explored antecedents of the search strategy while knowledge workers perform innovative activities. More importantly, by uncovering the dual temporal dynamics of innovative activities in new product development efforts, we provide an important managerial implication: Besides problem landscape (interdependencies) and locus of search, the optimal level of collaboration is determined, at least partly, by the two divergent temporal effects of problem-solving efforts.

The paper is organized as the following: Section two provides the theoretical arguments and hypotheses about the antecedents of the search strategy. Section three introduces our data and the methodology we used to test the hypotheses. Section four reports the results, and eventually, section five concludes the paper with discussions, boundary conditions of our findings, and directions for future research.

THEORY AND HYPOTHESES

Interdependencies and Search Strategy

Addressing the question of how organizations should manage activities across units and employees, organization theory literature highlights the particular importance of interdependencies (e.g., Galbraith, 1977; Thompson, 1967). More specifically, the pattern of interdependencies

between different elements of firms' knowledge base shapes the problem landscape of firms' future search endeavors. Especially, in knowledge-intensive contexts, experts with specialized knowledge of separate domains frequently make interdependent decisions (Puranam & Swamy, 2016). The cross-domain interdependencies, then, increase the likelihood of joint search, caused by agents' increased information processing needs, agents' bounded rationality in processing the interdependencies, and challenges of communication between experts from different domains.

First, the relevance of interdependencies to organizations' search strategy derives from the information-processing needs of the cooperating problem-solvers. As Galbraith (1977: 40) mentions, "in order to coordinate interdependent roles, organizations have invented mechanisms for collecting information, deciding, and disseminating information to resolve conflicts and guide interdependent actions." The more the internal interdependencies, the higher the task uncertainty, and the more intra-organizational information-processing required (Tushman & Nadler, 1978). Therefore, the need to acquire and efficiently manage interdependence-related information shapes the organizational search strategy (Aggarwal, Siggelkow, & Singh, 2011).

Second, bounded rationality (Cyert & March, 1963) is the theoretical underpinning of the information processing perspective. If agents were unboundedly rational, a single problem-solver could simply optimize across all possible configurations. However, considering the limited information-processing capability of bounded rational individuals, a higher level of collective search is required to address the information flows deriving from interdependencies. As extant research on organization design implies, information flow is a key factor in designing optimal organizations with different levels of decomposability (Burton & Obel, 1980; Simon, 1996; Stan & Puranam, 2016). Furthermore, the individuals' bounded rationality might hinder their complete

comprehension of the nature of the interdependencies between units (Adler, 1995; Gokpinar, Hopp, & Iravani, 2010; Sosa, Eppinger, & Rowles, 2004). In fact, inter-unit dependencies are inevitable in a world of imperfect decomposability, and they usually exist regardless of how well organizations partition tasks into units (Heath & Staudenmayer, 2000; Stan & Puranam, 2016).

Third, understanding the exact nature of the interdependencies between units might be problematic also because of the interaction impediments between specialists from separate units (Dougherty, 1992; Lawrence & Lorsch, 1967; Sosa, 2013; Stan & Puranam, 2016). Individuals' ability for cross-unit interactions is bounded, because of the issues such as knowledge specialization, different physical location, time pressure, and organizational boundaries (Sosa et al. 2004; Puranam & Swamy, 2016). Such communication challenges, in turn, complicate the management of boundary-spanning interdependencies even more. Concluding these arguments, hypothesis 1 follows:

> *Hypothesis 1: search over cross-domain interdependencies increases the likelihood of collective search.*

Locus of Search

In the presence of task interdependencies, the observed cooperative outcomes might be the result of actions taken by interdependent actors, thus making the connection between individual actions and outcomes problematic (March & Simon, 1958). This issue is particularly challenging when the interdependencies are novel and thus unknown to the agents, making it impossible to measure the individuals' contribution to the obscure observed outcome. Relying on such ambiguous feedback, including mixed information about one's own actions as well as others' input, increases the danger of superstitious learning by making conclusions from misleading observations

(Levinthal & March, 1993). Therefore, in the case of emerging new interdependencies, whose pattern is still unknown by the problem solvers, they need to participate in a series of mutual adjustment in order to find the new optimal combination of actions (Sosa et al., 2004; Stan & Puranam, 2016).

Furthermore, in distant search, where the interdependencies are emerging, and hence, less understood, knowledge workers rely on their own representations of the interdependencies. In such situations, problem-solvers cooperate based on their own mental representations, or mental models of their interdependencies (Rouse & Morris, 1986). For example, in a cooperative production task, the mental representation of engaging agents "delineates the impact of each individual's function and his or her contribution" to the collaborative outcome, and includes their beliefs about the way they must co-work to perform the task in hand (Converse, 1993; Puranam & Swamy, 2016).

Research shows that when experts engage in the joint search with their predisposed representations, then in most of the cases, the search performance is suboptimal. The reason is that when the experts engage in joint search with a mixture of good and bad representations of (newly formed) interdependencies, the likelihood of observing misleading feedback in the process increases, leading to the issue of superstitious learning, i.e. learning from misleading performance feedbacks (Levinthal and March 1993). Since the nature of newly formed interdependencies is still unknown, then, performance feedbacks of joint search activities are misleading and include mixed information about the value of unobserved actions taken by cooperating problem-solvers, because the realized outcome is jointly shaped by the agents' interdependent actions (Puranam & Swamy, 2016). Hence, we expect to observe less attempts of joint search while the knowledge workers explore distant loci.

Hypothesis 2: distant search (compared to local search) is less likely to be performed collectively.

Temporal Dynamics of Multistage Search

Complex problem landscapes, for example, as instantiated in the form of product architectures, tend to grow overly complex over time (Baldwin, MacCormack, & Rusnak, 2014; MacCormack, Rusnak, & Baldwin, 2007). Through the continuous search efforts, exploration of new domains, and annexing them to the firms' existing search landscape, the span and complexity of the firm's search space increases. Hence, knowledge workers require to constantly update their understanding of interdependencies and architecture of knowledge base because it will bind their future search efforts. Thus, in order to efficiently perform their problem-solving activities, and seamlessly embed the body of knowledge they create for solving a problem, the knowledge workers need to relentlessly update their understanding of the ever-changing architecture of knowledge and its interdependencies as shaped by other workers' activities. The knowledge workers, then, resort to more collaborative search, in the hope of keeping their architectural knowledge up-to-date. Thus, over time and within a given stage of the search, the need for collaboration will increase.

On the contrary, along with repeated search and problem-solving activities over longer periods of time and through subsequent stages of search, knowledge workers develop a common understanding of the firm's problem space, leading to gradual uncertainty resolution about the interdependencies in search activities. This, in turn, provides the opportunity for tacit coordination of the tasks (Srikanth & Puranam, 2011, 2014), reducing the explicit coordination requirements and collaboration needs over time.

Moreover, continuous cooperation with team members over repeated activities resolves uncertainties about the cooperative behavior of co-workers, diminishing the space for unidentifiable opportunistic behavior. This, in turn, reduces the epistemic interdependence between knowledge workers. Epistemic interdependence is the situation in which "one agent's optimal choices depend on a prediction of another agent's actions" (Puranam, Raveendran, & Knudsen, 2012, p. 420). Then, the developers who are epistemically interdependence need to collaborate while performing the dependent search efforts. However, epistemic interdependence diminishes when the agents' *predictive knowledge,* "the knowledge that enables one agent to act as though he or she can accurately predict another agent's actions" (Puranam et al., 2012, p. 420), improves as they continue to co-work with one another over long period of time. Moreover, the epistemic interdependence might reduce over time, as the accumulated knowledge about the interdependencies across repeated stages of search gradually undermines the effect of agent's bounded rationality in information processing; first, among set of all possible actions the experts begin to learn which are most influential to the search outcome, thus reducing the span of search. Second, they gradually learn about the performance outcome of the interdependencies, hence reducing the need to opt for collaborations to perform the search on a given known interdependence. Thus, over time and across subsequent stages of the search, the need for collaboration will decrease. The hypotheses 3 and 4 follow:

Hypothesis 3: As advancing through a given stage of the search, it is more likely to perform search collectively.

Hypothesis 4: As advancing across stages of the search, it is less likely to perform search collectively.

DATA AND METHODS

Research Setting

Utilizing a unique dataset, i.e. micro-level information on software architectures and engineers' collaboration patterns in software development, the study spans over a course of 60 months dedicated to the development of 13 versions of an industrial software package, collected from the software division of an HVAC (Heating, ventilation and air conditioning) technology premium supplier. The software is a plant supervision package—an integrated control solution that improves energy savings in HVAC systems. In 2015, the company reported revenues of 250 million euros with a 15% EBITDA margin, and is planning to go public soon.

In 2008, the software division started the systematic recording of development data (e. g. software architecture, development task assignments and task accomplishments data, etc.), while the development team was working on version 1.4 as the first full version of the software. Since then data have been stored in the company's digital repository. No data or incomplete data are available for earlier versions, also because, as confirmed by the chief software engineer in the interviews we conducted, version 1.4 of the analyzed software represented the first full-fledged version of the software and was a significant discontinuity in terms of software architecture and development activities. Since version 1.4, two incremental versions of the software were released internally (1.4.1 and 1.4.3), before the market release of version 1.5. In response to the competitors' release of new software applications with more user-friendly interfaces, the company decided to branch a new generation of its software with new functionalities (2.X branch, including 2.0.0, 2.0.1, 2.0.2, 2.0.3, and 2.1.0), while at the same time continuing to develop new versions of the software on the existing branch (1.X branch, including 1.5.2, 1.5.3, 1.5.4, and 1.5.5). These new versions

basically included improvements in the form of service packs (figure 1). Therefore, our dataset is divided into two parts, each capturing the data of one branch. The first dataset includes versions: 1.4.0, 1.4.1, 1.4.3, 1.5, 1.5.2, 1.5.3, 1.5.4, and 1.5.5 (called branch 1.X henceforth), while the second dataset keeps data of versions: 1.4.0, 1.4.1, 1.4.3, 1.5, 2.0.0, 2.0.1, 2.0.2, 2.0.3, 2.1.0 (called branch 2.X henceforth). We pooled the data of both branches into a single database, and thus the regression results reported in the finding section pertain to both branches. Nonetheless, separate analyses of the two branches demonstrate similar patterns and results. The following table reports a description of each version's characteristics.

Insert figure 1 about here.

Insert table 1 about here.

Multistage Search in the Software Development Context

The process of software development is not but an ongoing daily endeavor of solving problems. The team members engage in solving problems, either as minor as fixing a minuscule bug in functioning of a small module, or as major as creating novel function/modules and embedding them into the current software design, or even implementing more radical changes involving paradigm shift, e.g. using object-oriented paradigm in programming, or migrating to a new programming language.

Regarding the considerable variety of types of problems dealt with in software development, the search strategy to solve the problems shifts according to the problem-specific contingencies

involved. The problems in software development process are characterized by varying degree of complexity, span of search, and spread over the lengthy process of developing a given version, and long product lifecycle (as a software generally undertakes several gradual and radical generational changes) necessitating frequent oscillation between independent and joint search and problem-solving activities.

The problem landscapes might vary in degree of complexity. Some problems require search in least complex landscapes, involving changing single design element (i.e. file), whereas more complex problems include searching in landscapes characterized by interdependencies between several design elements. The most complex problems, then, need search over landscapes involving cross-domain interdependencies between design elements.

Problem-solving also might differ in the locus of search. Specifically, in software development context a problem may require searching locally by enhancing an existing solution (in the form of modifying current features and fixing bugs), or might demand distant search by creating novel solutions (in the form creating new features).

Lastly, as already discussed, the temporal dynamics of problem-solving in a multistage product development influences the shift in search strategy. The problems addressed later during a given stage of search (i.e. developing a given version of the software in our context) require more of joint search, compared to problems solved earlier during the same stage of search. On the contrary, problems that emerge later in the product lifecycle (i.e. in later versions of the software), ceteris paribus, are more likely to be solved via independent search efforts. Problem-specific parameters

inform the project leaders' decision as to whether set off independent individual developers or teams in solving a given problem in the context of software development.

Contextual Specificities of the Software Development

The context of our study, i.e. developing subsequent versions of an industrial software in a close-enterprise setting, has the following specific characteristics regarding the search and problem-solving perspective:

a. The generated knowledge, created by developers and through the problem-solving activities, is clustered around separate domains. Each developer, then, is knowledgeable about the domain assigned to him/her.
b. Over time, developers update their architectural knowledge of the software, as well as their understanding of other domains included in the software architecture and the interdependencies between their own domain and other domains. The updating process usually happens through repeated formal task coordination and collaborations, in addition to informal communications and knowledge sharing.
c. The problem landscape, including the software elements (files) and their interdependencies, continuously evolve and is endogenously shaped by the problem-solving activities of knowledge workers.

Data Description

In order to empirically investigate how *interdependencies, locus of search*, and *the temporal dynamics* of multistage search determine the shift in search strategy, from independent to joint search, we collected a unique design element-level dataset of an industrial software package, which

evolves over a period of 60 months and along 13 versions. The software is developed by the software division of a leading HVAC (heating, ventilation, and air conditioning) company based in Italy. The main function of the software is supervising HVAC micro-controllers in plants, such as industrial fridges and supermarkets.

Our dataset involves technical data of the architectural design of each software version, the problem landscape of firm's future search efforts, including the data of the design elements (i.e. files) and their interdependencies. In addition, we document the information of problem-solving activities performed on each design element during the development of each version, and the date and time of each activity. Further, we record the type of development task performed (creating new feature vs. modification of existing feature) on any given file, the developer who accomplished the task, and eventually, whether the task was performed independently by individual developers or jointly by a team of engineers.

Overall, this dataset provides an opportunity to examine daily search activities of knowledge workers and study how firms' search strategy shifts from independent to joint search, as well as to reveal the dynamics of multistage search and product development.

Measures

Unit of analysis: Each act of problem-solving in the context of software development includes changing and/or creating one or more files at once. The developers identify which files have to be changed to solve the problem at hand. Then, he/she downloads the files altogether into his/her own working station, makes the changes meant for solving the given problem in the downloaded files, saves the changes, and uploads the files back to the software repository. This process is called

making a *revision* in the software development setting. Hence, each *revision* involves a few micro-tasks on the files involved in the *revision*. For example, if in a given *revision* one file is changed, then the *revision* has one micro-task. If two files are changed, the *revision* includes two micro-tasks corresponding to each of the two files changed, and so on.

Therefore, the unit of our analyses is *micro-task*, the most basic building block of search efforts in our context. Each given *micro-task*, hence, pertains to a particular file involved in the *micro-task*. Consequently, each *revision* creates a few observations in our dataset, each of which represent a *micro-task* performed on one of the files changed in the given *revision*.

Dependent variable

Search strategy: It is the dependent variable and we measure it—at the level of *micro-task*—as whether the search effort is performed independently (then the dependent variable equals 0) or jointly (then the dependent variable is equal to 1) (as used in e.g. Aggarwal et al., 2011; Knudsen & Srikanth, 2014; Puranam & Swamy, 2016). A problem-solving is performed jointly when the developers co-work on the focal design element (file) in the given week.

Independent variables

Interdependencies: It captures the complexity of problem landscape, and is measured—at the level of *micro-task*—as the total number of cross-domain interdependencies a focal element (file) possesses at a given point in time, when the given problem-solving activity was performed. Cross-domain dependencies are identified as the dependencies between domains of search space, i.e., product modules, as derived from the application of the Louvain algorithm. We used the Louvain algorithm (Blondel, Guillaume, Lambiotte, & Lefebvre, 2008) to identify modules. This algorithm

is based on modularity optimization and is shown to outperform other known module-detection methods in terms of computational accuracy and time (Blondel et al., 2008).

Locus of search: As already mentioned, locus of search captures—at the level of *micro-task*—whether the focal search activity is local or distant search. Specifically, we measure if the focal problem-solving activity involves modifying current feature/file (i.e. local search, and the variables equals 0), or creating new feature/file (i.e. distant search, and the variable is equal to 1).

Within-stage progression: It records—at the level of *micro-task*—the lapse of time between the first week the division began developing the given *version* of the software and the week the focal problem-solving activity is carried out. We measure the variable in the number of weeks.

Across-stage progression: It documents—at the level of *micro-task*—the time interval between the first week the division began developing the software and the week the focal problem-solving activity is performed. We measure the variable in the number of weeks.

Control variables

Following common practice in the software engineering literature, we control for heterogeneity in the content of the files. Controlling for technical, file-related variation of the dependent variable is necessary to isolate the effect of the proposed explanatory variables from other possible sources of variation, namely from heterogeneity in the development activities deriving from file-specific characteristics.

The first control variable, Lines of Code, *LOC,* is a software metric used to measure the size of a computer program by counting the number of lines contained in the files' source code. In

addition, we account for the focal file's total number of interdependencies. Further, we use a binary variable, *Java language,* to control for the language in which the focal file is written (Java files=1, C and C header files=0). We also account for the *comment-to-code ratio*, measuring the possible explicit knowledge sharing via lines of instructions contained within the focal file. Eventually, we control if the developer who accomplished the focal problem-solving activity is the owner of the focal file (the owner of a file is the developer who first created the file. In the subsequent versions, the owner is the developer who accomplishes most of the changes (if any) on the focal file.). We expect to observe less of joint search in cases where the developer and owner are the same.

Model specification

To test our hypotheses, we use logit regressions, where the unit of analysis, denoted as *i*, is the micro-tasks, the building blocks of search and problem-solving activities. We estimate the effect of independent and control variables, with the software versions' and developers' fixed effects, in order to (partially) control for potential sources of unobservable variation in the dependent variable. Thus, we estimate the following regression equation:

$$\begin{aligned} Search\ strategy_i = {} & \beta_0 + \beta_1 \times Interdependences_i + \beta_2 \times Locus\ of\ search_i \\ & + \beta_3 \times Within - stage\ progression_i + \beta_4 \times Across - stage\ progression_i \\ & + X_{control} B_5 + \tau_t + \lambda_d + \varepsilon_i \end{aligned}$$

where the dependent variable is search strategy, capturing the independent versus collaborative nature of the search activity *i*. The independent variables are: a) *interdependencies*, measured as the number of cross-module interdependencies the focal file that is involved in the search activity *i* has; b) *locus of search*, measured as whether the search activity *i* involves modifying current files

or entails creating new files; c) *within-stage progression*, measured as the week in which the focal search activity *i* is carried out, with reference to the first week of developing current software *version*; and d) *across-stage progression*, measured as the week in which the focal search activity *i* is carried out, with reference to the first week of developing the studied *software*. In addition, we control for the technical characteristics of each focal file involved in the search activity *i*, and we use different model specifications which include software versions' fixed effects τ_t and developers' fixed effect λ_d.

As an alternative approach, and to control for potential issue of auto-regression, we exploited a peculiar characteristic of our data, creating and estimating a diff-in-diff model. As mentioned in the previous section, after developing 4 versions of the software, the marketing department of the analyzed company requested that the software division produce a radically novel version of the software, with redesigned user interfaces and many other new features. This request was fully exogenous to the software division strategy and organization, as it was forced by the competitors' launch of new and innovative versions of their software. At the same time, the software division continued to develop improvements and service packs for the old version in order to keep serving customers who chose not to upgrade. Therefore, in our setup we have two sub-datasets and leverage on the exogenous shock in order to better identify our mechanisms via a diff-in-diff approach. The new branch includes data for the software versions 2.0.0 to 2.1.0 (2.X), and represents the "treated" group of files, as created in response to the exogenous shock. The "control" group of files includes the software versions developed in the old branch, i.e. 1.5.2 to 1.5.5 (1.5.X).

FINDINGS

Descriptive Statistics and Correlations

Table 2 reports the descriptive statistics of the variables. We observe 11,855 *micro-tasks* per file performed while developing the 13 versions of the software. Of all the micro-tasks accomplished for solving problems, about 9% are carried out jointly, confirming the fact that the studied software division manages the product development activities majorly through independent individual task performances. Moreover, each design element that was changed in a micro-task, on average, has 16 interdependencies, of which 6 are cross-module (37%). Please note that in our analyses we only capture the files that are being changed in micro-tasks during the development of software versions. Thus, we expect to record observations of files with higher cross-module interdependencies because they usually require more work and iterations of design activities.

In addition, in about one third of search efforts the problem-solvers explored distant loci, whereas 68% of the problem-solving activities were performed locally, through modifications of existing design elements. This fact indicates that while the managers focused on continuous improving of the existing features' performance, at the same time, they always maintained a remarkable amount of exploratory innovation on their product. Overall, we capture 238 weeks of developing the studied software versions, the most challenging of which took 68 weeks of development work (version 2.0.0).

In addition, on average the files contain 334 lines of code, whose distribution is skewed towards the left, as expected. Further, in our sample, for every three lines of code contained in file there is one line of comment, usually containing the guidance and/or instructions about the

functionality of a particular part of the code. It appears that the project leaders could manage documenting and explicating the specific knowledge about internal content of files, facilitating the transfer of knowledge to other developers for their future reference, and hence, reducing the need for future potential collaborations. Eventually, in 65% of the micro-task performances the developer was the owner of the focal file involved in the micro-tasks, indicating that, as anticipated, the majority of the search efforts was conducted within the knowledge domain of developers.

Insert table 2 about here.

Table 2 also reports pair-wise correlation estimates between the main variables of the study. The sign and significance of the correlation coefficients seem to point in the direction of our hypotheses. However, the size of the correlation coefficients between the main variables is small enough to reduce the concern of collinearity. The *locus of search* has the significant and considerable correlation with *within-* and *across-stage progression*: over time, we observe less exploratory search, i.e., fewer creations of new features, and instead, more modifications of files. This is consistent with our findings from field visits, during which we learned that the managers begin the development of each version with creating new features, and later focus on modifying them, as well as, improving the functioning of existing features inherited from the past versions.

Hypothesis Testing

Table 3 reports logit estimation results regarding the study's four hypotheses. The first model (column 1) reports the estimations of the base model including control variables. In the second

model we add the independent variables. The estimates find support for our hypotheses. Cross-domain interdependencies increase the probability of shifting from independent to joint problem-solving. Further, distant search is more likely to be performed independently. Regarding the dual effect of time, the estimates suggest that later during the development of a given version the problem-solvers perform the searches jointly, whereas over the course of developing the software across subsequent versions developers gradually restrain from joint problem-solving.

The third model adds the software versions' fixed effects in order to account for possible systematic differences between versions. In fact, this is a necessary check, because in the field interviews conducted with the developers and managers we discovered that some of the software versions included more radical changes and new feature developments, while others were more incremental, basically entailing modifications of features inherited from previous versions. The patterns of the relationships—even after accounting for version specificities—replicates those of the previous model, supporting hypotheses 1 to 4. In the fourth model, we add developers' fixed effects. Again, our hypotheses are supported. The last model (column 5 in table 3) includes both versions' and developers' fixed effect, wherein we found significant support for the four hypotheses. The results also show continuous improvement of explanatory power of the subsequent models.

In addition, regarding the effect of control variables, the results demonstrate that *lines of code,* as well as *total number of interdependencies* a focal file holds significantly increase the probability of shifting from independent to joint search strategy. Finally, as expected, if the developer of a given micro-task is the owner of the focal file, then it is more likely that the problem-solving is performed individually.

Insert table 3 about here.

Differences-in-Differences Analysis

As previously discussed, in order to react to competitors' innovations, the marketing department of the analyzed company requested the software division to radically redesign the software at a certain point in time. Thanks to this shock, which we can consider exogenous to the software division, we partition our dataset into two sub-datasets. The first dataset is for the old, existing software versions and development activities, including 1.5.X versions—which continues to be developed as service packs, and modification releases for existing, non-upgrading customers. The second sub-dataset is for the new software versions and development activities (2.X). We set up a quasi-experimental design in which the new trajectory of software design evolution will be impacted (or treated) by the exogenous decision of redesign, whereas the releases of service packs for the earlier version are regarded as the control group.

Table 4 below reports the results of our diff-in-diff analysis. Model 1 includes the direct treatment effect and demonstrates the significant systematic shift towards joint problem-solving in the software versions developed after the exogenous shock of redesigning the software. However, beyond the post-shock effect we do not find a significant difference between the search strategy implemented in new versions (2.X) and the versions developed on the old branch (1.5.X). This finding, which is replicated in models 2 and 3 in table 4, including control and independent variables, corroborates our previous results. As suggested by the diff-in-diff results, no matter

whether a product architecture is inherited from the past or reshuffled by an exogenous shock, our main hypotheses are supported. This finding suggest that our previous results are valid also when the issue of auto-regression is mitigated as a result of an exogenous radical change in the problem landscape. Nonetheless, it is evident that after the software redesign, on average, search efforts are pursued jointly rather than independently, probably because the overall software complexity increased after the redesign.

Insert table 4 about here.

Robustness Checks

As shown earlier in table 3, our findings are robust to a number of alternative model specifications. The model including developers' fixed-effects delivers identical results. Adding software versions' fixed effects, and then both versions' and developers' fixed effect, the results also hold. As an additional robustness check, we re-ran the regressions using a probit specification. The findings are similar to previous results. The estimation of probit models can be found in Appendix 1.

Our findings are also similar when we used an alternative operationalization of the dependent variable. In the previous analyses, to measure the search strategy we defined joint search as co-working on a focal file in a given week. In an alternative operationalization, in order to identify joint problem-solving, we observe whether the co-working occurred at a focal directory enclosing the focal file in a given week. We found comparable support for the hypotheses using this alternative definition of the dependent variable. Results can be found in appendix 2.

Moreover, we estimated the models on two different sub-samples, one encompassing software versions developed for branch 1.X (1.4 to 1.5.5) and the other including the new branch 2.X (1.4, 1.4.1, 1.4.3, 1.5, 2.0.0, 2.0.1, 2.0.2, 2.0.3, 2.1.0). The findings are similar to what we found previously, providing more support for the hypotheses. Results pertaining to each of the two branches could be found in appendices 3 and 4.

DISCUSSION AND CONCLUSION

This study examines the determinants of search strategy in knowledge-intensive firms. It investigates how problems' characteristics affect the managerial decision of assigning independent individuals versus teams to solving given problems, and uncover the temporal dynamics of such relationship. Benefiting from a unique dataset of micro-level technical and organizational data on multistage search activities during the development of an industrial software, we document how cross-domain interdependencies and locus of search, as well as progressing through the development phase result in organizations' shift from independent to collaborative search.

We apply a variety of regression techniques in order to test the hypotheses and find that cross-module interdependencies increase the likelihood of resorting to collaborative problem-solving. On the contrary, it is less likely to opt to joint search while performing distant search, in the form of creating new solutions/features, whereas local search, in the form of modifying current features, is pursued jointly more often. Eventually, we document two seemingly conflicting effect of time on the problem-solving strategy during the development of the studied software. Approaching towards the end of given stage of search, i.e. the time to release the given version of the software, we observe that developers turn to joint search for solving their problems. However, over the course

of developing subsequent versions of the software the developers gradually opt to independent search and problem-solving.

Our findings complement the existing search and innovation literature in two ways. This research is among the first attempts to relax a widely-used but constraining assumption in simulation studies examining organizational search, i.e. exogeneity of problem landscape. Most of the studies so far assume that the organizations' problem landscape is exogenous to the firms that are randomly distributed on the landscape. Our processual dataset provides the opportunity to examine the determinants of search strategies when the organizations' problem landscape is endogenously formed by the problem-solving activities of the firms' knowledge workers. During each stage of search (developing a given software version), the knowledge workers (developers) devise new solutions (create new features) and embed them back into the existing problem landscape (product architecture), adding new elements and forming new interdependencies in the current problem landscape forming the basis for the organization's future search activities.

Another key contribution of this study is introducing the dual divergent effect of time on the organizations' search strategy. The results reveal that over the course of developing the software's subsequent versions, the developers turn to independent search. However, during the development of each version the developers resort to joint searches. This finding reconciles the ostensibly conflicting arguments in the general debate of organization of innovation, where one stream of research argues for increasing complexity over time, necessitating more of collaborative search, while other studies argue for generation of a common understanding (of the product architecture) among the knowledge workers, reducing the need for joint search. We shed light on this debate by

disentangling dual dynamics of multistage search, within- and across-stages, and uncovering the mechanisms that differently shape the organizations' search strategy at different points in time.

Several limitations apply to this study. First, we examined organizational decisions and temporal changes in the context of a specific company, and in the software industry. While this context is replete with frequent problem-solving activities on a daily basis and ideal for studying organizations' search efforts, the generalizability of our findings to other contexts where product components are more complex, and problem-solving activities are different in nature and intensity has to be tested. Second, we studied a commercial software package developed in a closed enterprise setting. Therefore, it is likely that our findings do not fully apply to the context of open-source software development, or similar contexts of voluntary problem-solving and open organizations. Third, as already shown, most of the innovative activities studied in this research were incremental (local search) in nature rather than radical (distant search). Our suggested implications should be reexamined for cases of radical innovations, as the company's existing problem landscape is disrupted, and therefore the antecedents of the search strategy should be different.

This study opens some avenues for future research in search and innovation literature. First, building upon our conceptualization and operationalization of technological design and change at the file-level, and of the organizational structure at the micro-task level, future studies could uncover other underlying mechanisms that are involved in the problem-solving and search endeavors in organizations. New insights will inform key managerial decisions in the fairly overlooked contexts that are characterized by turbulent environments replete with technological changes. Furthermore, future studies might examine the organizational contingencies of the dual

temporal dynamics of inter-generational innovation discussed here, and shed more light on driving factors of each of the two conflicting dynamics. Moreover, in order to replicate and validate our findings in contexts other than the software industry, future research has to look into other innovation-intensive industries, uncover new contingencies, and provide new insights.

FIGURES AND TABLES

Figure 1. Progression of software versions over the period of study

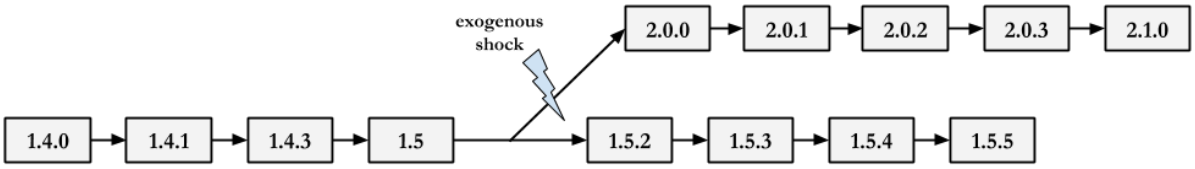


Table 1. Descriptive data of the analyzed software versions and the corresponding development process

Version	**1.4**	**1.4.1**	**1.4.3**	**1.5**	**1.5.2**	**1.5.3**	**1.5.4**	**1.5.5**	**2.0.0**	**2.0.1**	**2.0.2**	**2.0.3**	**2.1.0**
Release Dates	12/15/08	3/17/09	4/29/09	7/27/09	3/8/10	11/3/10	2/17/11	3/7/12	11/30/10	5/16/11	10/10/11	7/23/12	3/28/13
# of design elements (i.e. files)	3152	3153	3153	3416	3603	3613	3616	3795	3868	3960	4014	4030	4149
# of micro-tasks per file	2800	207	192	803	900	1346	251	917	2531	324	421	562	601
# of interdependencies b/w files	14887	14932	14946	18838	20032	20047	20061	20784	21067	21482	21774	22156	22914
# of developers	6	9	8	11	11	13	10	8	15	10	9	7	8

Table 2. Pairwise correlation results

	Mean	S.D.	Min	Max	1	2	3	4	5	6	7	8	9	10
1. Search strategy	0.09	0.28	0	1	1									
2. Number of cross-domain interdependencies	6.22	7.23	0	38	0.16	1								
3. Locus of search	0.32	0.47	0	1	-0.2	*-0.02*	1							
4. Within-stage progression	17.23	16.47	0	68	0.09	0.03	-0.34	1						
5. Across-stage progression	82.25	63.79	0	238	0.06	*-0.01*	-0.48	0.46	1					
6. Lines of code/100	3.34	4.37	0	36.72	0.28	0.32	-0.34	0.16	0.19	1				
7. Total number of interdependencies	15.93	12.47	1	65	0.14	0.85	*-0.02*	0.03	0.05	0.45	1			
8. Java files	0.87	0.33	0	1	-0.03	0.38	0.11	-0.07	-0.12	-0.13	0.23	1		
9. Comment-to-code ratio	0.29	1.84	0	64	-0.02	-0.05	0.04	-0.02	-0.02	-0.07	-0.03	-0.15	1	
10. Owner-developer	0.65	0.48	0	1	-0.15	-0.2	0.2	-0.11	*0*	-0.13	-0.16	-0.2	0.06	1

Total number of observations is 11,855. All correlation coefficients, except for those in Italic, are significant with 95% confidence interval or higher.

Table 3. Logit regression results testing the main hypotheses

Dependent variable *Search strategy*	(1)	(2)	(3)	(4)	(5)
Cross-domain Interdependencies		0.039***	0.030***	0.044***	0.034***
		(0.01)	(0.01)	(0.01)	(0.01)
Locus of search		-2.280***	-2.237***	-2.170***	-2.587***
		(0.22)	(0.25)	(0.24)	(0.28)
Within-stage progression		0.007**	0.008***	0.020***	0.027***
		(0.00)	(0.00)	(0.00)	(0.00)
Across-stage progression	-0.005***	-0.010***	-0.009***	-0.016***	-0.020***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code/100	0.080***	0.078***	0.044***	0.086***	0.046***
	(0.01)	(0.01)	(0.01)	(0.01)	(0.01)
Total interdependencies	0.027***	0.000	0.014*	-0.003	0.012*
	(0.00)	(0.01)	(0.01)	(0.01)	(0.01)
Java files	-0.525***	-0.857***	-0.344	-0.685***	-0.340
	(0.12)	(0.13)	(0.19)	(0.14)	(0.18)
Comment-to-code ratio	-0.077	-0.035	-0.003	-0.016	0.008
	(0.06)	(0.03)	(0.02)	(0.02)	(0.02)
Developer-owner	-0.756***	-0.678***	-0.727***	-0.654***	-0.703***
	(0.08)	(0.08)	(0.08)	(0.08)	(0.09)
Constant	-1.792***	-0.947***	-1.476***	-1.222***	-1.041*
	(0.15)	(0.17)	(0.25)	(0.21)	(0.46)
Versions' fixed effect	-	-	Y	-	Y
Developers' fixed effects	-	-	-	Y	Y
Wald Chi^2	715.48***	781.99***	843.40***	1052.14***	1094.93***
Log pseudo-likelihood	-2945.618	-2828.179	-2763.546	-2699.721	-2634.571
Pseudo R^2	0.116	0.151	0.170	0.190	0.209
Number of observations	9604	9604	9604	9604	9604

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

Table 4. Logit regression results testing the difference-in-differences model

Dependent variable *Search strategy*	(1)	(2)	(3)
Post version 1.5	1.216***	0.311*	0.363*
	(0.10)	(0.14)	(0.15)
Treatment X post version 1.5	-0.064	-0.108	-0.262
	(0.14)	(0.16)	(0.17)
Cross-domain Interdependencies			0.027***
			(0.01)
Locus of search			-2.007***
			(0.24)
Within-stage progression			0.012***
			(0.00)
Across-stage progression		-0.007***	-0.008***
		(0.00)	(0.00)
Lines of Code/100		0.045***	0.047***
		(0.01)	(0.01)
Total number of interdependencies		0.033***	0.014**
		(0.00)	(0.01)
Java files		-0.034	-0.337
		(0.17)	(0.18)
Comment-to-code ratio		-0.021	-0.003
		(0.03)	(0.02)
Developer-owner		-0.715***	-0.716***
		(0.08)	(0.08)
Constant	-3.267***	-2.855***	-2.439***
	(0.08)	(0.38)	(0.40)
Wald Chi^2	289.392***	1094.572***	1085.139***
Log pseudo-likelihood	-3987.365	-2818.603	-2747.373
Pseudo R^2	0.040	0.167	0.185
Number of observations	15857	10013	9962

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

APPENDICES

Appendix 1. Probit regression results

Dependent variable *Search strategy*	(1)	(2)	(3)	(4)	(5)
Cross-domain Interdependencies		0.018***	0.021***	0.010*	0.013**
		(0.00)	(0.00)	(0.00)	(0.00)
Locus of search		-1.046***	-0.984***	-0.995***	-1.117***
		(0.12)	(0.13)	(0.14)	(0.18)
Across-stage progression		-0.005***	-0.007***	-0.004***	-0.009***
		(0.00)	(0.00)	(0.00)	(0.00)
Within-stage progression	0.002	0.004**	0.011***	0.004**	0.014***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code/100	0.040***	0.042***	0.045***	0.022***	0.024***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Total number of interdependencies	0.014***	0.002	0.001	0.011***	0.010**
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Java files	-0.299***	-0.427***	-0.349***	-0.084	-0.059
	(0.06)	(0.07)	(0.07)	(0.09)	(0.09)
Comment-to-code ratio	-0.018	-0.014	-0.006	-0.001	0.006
	(0.02)	(0.01)	(0.01)	(0.01)	(0.01)
Developer-owner	-0.428***	-0.370***	-0.358***	-0.380***	-0.363***
	(0.04)	(0.04)	(0.04)	(0.05)	(0.05)
Constant	-1.289***	-0.676***	-0.810***	-1.073***	-0.847**
	(0.07)	(0.09)	(0.13)	(0.14)	(0.26)
Versions' fixed effect	-	-	Y	-	Y
Developers' fixed effects	-	-	-	Y	Y
Wald Chi^2	631.960***	681.941***	751.328***	955.092***	995.083***
Log pseudo-likelihood	-2568.139	-2471.779	-2410.307	-2365.326	-2307.754
Pseudo R^2	0.116	0.149	0.170	0.186	0.206
Number of observations	8281.000	8281.000	8281.000	8281.000	8281.000

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

Appendix 2. Probit regression results for the alternative operationalization of the dependent variable

Dependent variable *Search strategy* *(alternative operationalization)*	(1)	(2)	(3)	(4)	(5)
Cross-domain Interdependencies		0.019***	0.022***	0.010*	0.013**
		(0.00)	(0.00)	(0.00)	(0.00)
Locus of search		-0.738***	-0.790***	-0.719***	-0.831***
		(0.06)	(0.07)	(0.08)	(0.09)
Across-stage progression		-0.004***	-0.005***	-0.005***	-0.007***
		(0.00)	(0.00)	(0.00)	(0.00)
Within-stage progression	-0.000	0.002	0.005**	0.001	0.006***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code/100	0.027***	0.029***	0.030***	0.011**	0.011*
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Total interdependencies	0.017***	0.005	0.003	0.012***	0.011***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Java files	0.301***	0.194***	0.250***	0.655***	0.686***
	(0.05)	(0.06)	(0.06)	(0.07)	(0.07)
Comment-to-code ratio	-0.082*	-0.058	-0.044	-0.042	-0.022
	(0.04)	(0.03)	(0.03)	(0.03)	(0.02)
Developer-owner	-0.617***	-0.557***	-0.541***	-0.618***	-0.613***
	(0.03)	(0.03)	(0.03)	(0.03)	(0.03)
Constant	-0.845***	-0.300***	-0.286**	-0.719***	-0.065
	(0.06)	(0.07)	(0.10)	(0.11)	(0.17)
Versions' fixed effect	-	-	Y	-	Y
Developers' fixed effects	-	-	-	Y	Y
Wald Chi^2	1175.62***	1431.15***	1581.73***	1833.30***	1975.52***
Log pseudo-likelihood	-4561.933	-4423.330	-4357.008	-4276.512	-4197.368
Pseudo R^2	0.126	0.153	0.166	0.181	0.196
Number of observations	8439	8439	8439	8439	8439

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

Appendix 3. Logit regression results on the subsample of branch 1.X

Dependent variable *Search strategy*	(1)	(2)	(3)	(4)	(5)
Cross-domain Interdependencies		0.051***	0.054***	0.041***	0.045***
		(0.01)	(0.01)	(0.01)	(0.01)
Locus of search		-2.141***	-2.140***	-2.136***	-2.553***
		(0.32)	(0.35)	(0.36)	(0.41)
Within-stage progression		-0.011*	0.000	-0.002	0.015*
		(0.01)	(0.01)	(0.01)	(0.01)
Across-stage progression	-0.003**	-0.005***	-0.009***	-0.004*	-0.014***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code/100	0.112***	0.112***	0.118***	0.065***	0.061***
	(0.02)	(0.01)	(0.01)	(0.02)	(0.02)
Total interdependencies	0.024***	-0.008	-0.009	0.007	0.008
	(0.00)	(0.01)	(0.01)	(0.01)	(0.01)
Java files	-0.550***	-0.948***	-0.794***	-0.502	-0.460
	(0.16)	(0.18)	(0.18)	(0.27)	(0.26)
Comment-to-code ratio	-0.158	-0.078	-0.034	-0.022	-0.003
	(0.08)	(0.05)	(0.02)	(0.02)	(0.02)
Developer-owner	-0.723***	-0.655***	-0.612***	-0.730***	-0.672***
	(0.11)	(0.11)	(0.11)	(0.12)	(0.13)
Constant	-1.985***	-1.051***	-1.172***	-1.479***	-1.275
	(0.21)	(0.25)	(0.30)	(0.36)	(0.69)
Versions' fixed effect	-	-	Y	-	Y
Developers' fixed effects	-	-	-	Y	Y
Wald Chi^2	358.15***	370.01***	436.84***	577.69***	628.21***
Log pseudo-likelihood	-1435.900	-1382.283	-1342.613	-1307.648	-1267.614
Pseudo R^2	0.114	0.147	0.171	0.193	0.218
Number of observations	4596	4596	4596	4596	4596

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

Appendix 4. Logit regression results on the subsample of branch 2.X

Dependent variable *Search strategy*	(1)	(2)	(3)	(4)	(5)
Cross-domain Interdependencies		0.034**	0.038***	0.026*	0.029**
		(0.01)	(0.01)	(0.01)	(0.01)
Locus of search		-2.244***	-2.163***	-2.253***	-2.564***
		(0.31)	(0.33)	(0.34)	(0.38)
Within-stage progression		0.015***	0.027***	0.015***	0.031***
		(0.00)	(0.00)	(0.00)	(0.01)
Across-stage progression	-0.007***	-0.012***	-0.016***	-0.012***	-0.020***
	(0.00)	(0.00)	(0.00)	(0.00)	(0.00)
Lines of Code/100	0.080***	0.078***	0.079***	0.052***	0.048***
	(0.01)	(0.01)	(0.01)	(0.01)	(0.01)
Total interdependencies	0.023***	-0.001	-0.003	0.011	0.011
	(0.00)	(0.01)	(0.01)	(0.01)	(0.01)
Java files	-0.343	-0.562**	-0.522*	-0.128	-0.221
	(0.19)	(0.20)	(0.21)	(0.25)	(0.26)
Comment-to-code ratio	-0.033	-0.009	-0.002	0.010	0.015
	(0.05)	(0.03)	(0.02)	(0.02)	(0.02)
Developer-owner	-0.765***	-0.649***	-0.675***	-0.697***	-0.722***
	(0.11)	(0.11)	(0.11)	(0.12)	(0.12)
Constant	-1.824***	-1.207***	-1.364***	-1.608***	-0.800
	(0.21)	(0.25)	(0.30)	(0.35)	(0.62)
Versions' fixed effect	-	-	Y	-	Y
Developers' fixed effects	-	-	-	Y	Y
Wald Chi^2	408.38***	434.69***	432.43***	522.30***	518.54***
Log pseudo-likelihood	-1498.036	-1423.821	-1410.341	-1374.738	-1357.391
Pseudo R^2	0.124	0.168	0.175	0.196	0.206
Number of observations	5008	5008	5008	5008	5008

*Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1*

REFERENCE

Adler, P. S. 1995. Interdepartmental interdependence and coordination: The case of the design/manufacturing interface. ***Organization science***, 6(2): 147–167.

Aggarwal, V. A., Siggelkow, N., & Singh, H. 2011. Governing collaborative activity: interdependence and the impact of coordination and exploration. ***Strategic Management Journal***, 32(October 2008): 705–730.

Baldwin, C., MacCormack, A., & Rusnak, J. 2014. Hidden structure: Using network methods to map system architecture. ***Research Policy***, 43(8): 1381–1397.

Baumann, O., & Siggelkow, N. 2013. Dealing with Complexity: Integrated vs. Chunky Search Processes. ***Organization Science***, 24(1): 116–132.

Blondel, V. D., Guillaume, J.L., Lambiotte, R., & Lefebvre, E. 2008. Fast unfolding of communities in large networks. ***Journal of Statistical Mechanics: Theory and Experiment***, 2008(10): 6.

Burton, R. M., & Obel, B. 1980. A computer simulation test of the M-form hypothesis. ***Administrative Science Quarterly***, 457–466.

Converse, S. 1993. Shared mental models in expert team decision making. ***Individual and group decision making: Current***, (1993): 221.

Cross, R., Rebele, R., & Grant, A. 2016. Collaborative Overload. ***Harvard Business Review***, 94(1–2): 74–79.

Cyert, R. M., & March, J. G. 1963. ***A behavioural theory of the firm***.

Dougherty, D. 1992. Interpretive barriers to successful product innovation in large firms. ***Organization science***, 3(2): 179–202.

Galbraith, J. R. 1977. ***Organization design***. Addison Wesley Publishing Company.

Gavetti, G., Helfat, C. E., & Marengo, L. 2016. ***Shaping, searching, and endogenous selection: The quest for superior performance***. Working paper.

Gokpinar, B., Hopp, W. J., & Iravani, S. M. R. 2010. The impact of misalignment of organizational structure and product architecture on quality in complex product development. ***Management Science***, 56(3): 468–484.

Heath, C., & Staudenmayer, N. 2000. Coordination neglect: How lay theories of organizing complicate coordination in organizations. ***Research in organizational behavior***, 22: 153–191.

Knudsen, T., & Srikanth, K. 2014. Coordinated Exploration: Organizing Joint Search by Multiple Specialists to Overcome Mutual Confusion and Joint Myopia. ***Administrative Science Quarterly***, 59(3): 409–441.

Lawrence, P. R., & Lorsch, J. W. 1967. ***Organization and environment: Managing differentiation and integration***. Division of Research, Graduate School of Business Administration, Harvard University Boston, MA.

Levinthal, D. A., & March, J. G. 1993. The myopia of learning. ***Strategic Management Journal***, 14: 95–112.

MacCormack, A., Rusnak, J., & Baldwin, C. 2007. The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry. ***Harvard Business School***

Working Paper.

March, J. G., & Simon, H. A. 1958. Organizations.

Nonaka, I. 1994. A dynamic theory of organizational knowledge creation. ***Organization science***, 5(1): 14–37.

Puranam, P., Raveendran, M., & Knudsen, T. 2012. Organization Design: The Epistemic Interdependence Perspective. ***Academy of Management Review***, 37(3): 419–440.

Puranam, P., & Swamy, M. 2016. How Initial Representations Shape Coupled Learning Processes. ***Organization Science***, 27(2): 323–335.

Rouse, W. B., & Morris, N. M. 1986. On looking into the black box: Prospects and limits in the search for mental models. ***Psychological bulletin***, 100(3): 349.

Simon, H. A. 1996. ***The sciences of the artificial***. MIT press.

Sosa, M. E. 2013. Realizing the Need for Rework: From Task Interdependence to Social Networks. ***Production and Operations Management***, 23(8): 1312–1331.

Sosa, M. E., Eppinger, S. D., & Rowles, C. M. 2004. The Misalignment of Product Architecture and Organizational Structure in Complex Product Development. ***Management Science***, 50(12): 1674–1689.

Srikanth, K., & Puranam, P. 2011. Integrating distributed work: Comparing task design, communication, and tacit coordination mechanisms. ***Strategic Management Journal***, 32(February): 849–875.

Srikanth, K., & Puranam, P. 2014. The Firm as a Coordination System: Evidence from Software Services Offshoring. ***Organization Science***, (February).

Stan, M., & Puranam, P. 2016. Organizational adaptation to interdependence shifts: The role of integrator structures. ***Strategic Management Journal***.

Thompson, J. D. 1967. ***Organizations in action: Social science bases of administrative theory***. Transaction Publishers.

Tushman, M. L., & Nadler, D. A. 1978. Information Processing as an Integrating Concept in Organizational Design. ***Academy of management review***, 3(3): 613–624.

ENDNOTES

[i] We wish to emphasize that the use of the term "modularity" in the management and engineering literatures does not refer to mere commonality (i.e. when multiple products share some common components) nor combinability (i.e. different components can be combined into an end product configuration). Any compound product might have these two characteristics. For example, a pharmaceutical company might produce multiple products that utilize a common binding agent (commonality), and by adding particular ingredients to a product the company might augment the product's function (combinability). Product modularity here indicates a higher degree of flexibility and indeterminateness of form and function. That is, components, and the end products they will eventually compose, are designed in such a way that they retain the capacity to be re-combined into new configurations.

[ii] Indeed, this is an overlapping point between Baldwin and Clark's (2000) and Ulrich's (1995) technical definitions of modularity. Ulrich (1995) recognizes "…the phenomenon of a single component implementing several functional elements [that] is called function sharing" (p. 423). He asserts that a "…modular architecture allows the required changes that are typically associated with the product's function to be localized to the minimum possible number of components" (p. 427). As a consequence, he implies that the number of (shared) functions implemented by a component positively co-varies with the number of cross-component interfaces.

[iii] Understand is a static code analysis tool, an IDE built to help software designers and architects comprehend their source code, analyze it, measure it, and maintain it. The software

helps engineers understand their code by identifying the dependencies in the codes and reporting how they connect. It also provides information about functions, classes, variables, etc., as well as how they are used, called, modified, and interacted with.

[iv] Please note that in our analyses we cannot use panel data regressions. The reason is that multiple tasks might be performed on the same dependency dyad between two given files during the development of a given software version. Nonetheless, we estimate versions' and developers' fixed effects to control for potential sources of unobservable variation in the dependent variable.