Path Dependence and Organizational Choices in Product Architecture Evolution

Mahdi Ebrahim
Maastricht University
School of Business and Economics
mahdi.ebrahim@gmail.com

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

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.
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
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 desired effect of design modularity on containing future product complexity is stronger when development tasks are assigned to 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*¹.

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

¹ Therefore, we consider cyclacity as the inverse of hierarchy.
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 desired effect of design hierarchy on containing future product complexity is stronger when development tasks are assigned to teams (rather than individuals).

**Methodology**

A complex product comprises a large number of elements, each of which accomplishes a specific function. The proper functioning of a complex 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 below 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

---

2 For example, when element $A$ depends on element $B$ to accomplish a task, then, one could assume a dependency between two elements that take the form of a directional link connecting element $A$ to $B$. 
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\(^3\), 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.

\(^3\) This data has been extracted by the means of 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 $\delta$-function $\delta(u, v)$ is 1 if $u = v$ and 0 otherwise, and $m = \frac{1}{2} \sum_{i,j} A_{ij}$ (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\textsuperscript{4}. The descriptive statistics of the dependent and independent variables are reported in table 2 in the results section.

\textit{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, \textit{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, \textit{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, \textit{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.

\textbf{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:

\[
\text{design complexity}_{it} = \beta_0 + \beta_1 \text{modularity}_{it-1} + \beta_2 \text{hierarchy}_{it-1} + X_{it-1}B + \mu_i + \epsilon_{it}
\]

\textsuperscript{4} For a detailed review of the methodology and proofs, please refer to the original paper: Baldwin et al., 2014.
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:

\[
\text{design complexity}_{it} = \beta_0 + \beta_1 \text{modularity}_{it-1} + \beta_2 \text{hierarchy}_{it-1} \\
+ \gamma_1 \text{modularity}_{it-1} \times \text{task assignment}_{it'} + \gamma_2 \text{hierarchy}_{it-1} \times \text{task assignment}_{it'} \\
+ X_{it-1}B + \tau_t + \lambda_d + \varepsilon_{it}
\]
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 \( t-1 < t' < t \)). 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 \( \tau_t \) and developers’ fixed effect \( \lambda_d \).

**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\( \times \)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 more complex, over time, than
versions 1.X. Table 1 includes a wider set of descriptive statistics about the analyzed
software versions.

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.

Table 3 reports the correlation coefficients between the variables entered into 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.

Regression results for path dependence in 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.

Regression results for task assignment and performance moderation effects

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 cyclinality detrimentally increases file’s future complexity.

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 up 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.
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

Number of Files

Number of Direct Dependencies

Number of Indirect Dependencies
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>
<thead>
<tr>
<th>Version</th>
<th>1.4</th>
<th>1.4.1</th>
<th>1.4.3</th>
<th>1.5</th>
<th>1.5.2</th>
<th>1.5.3</th>
<th>1.5.4</th>
<th>1.5.5</th>
<th>2.0.0</th>
<th>2.0.1</th>
<th>2.0.2</th>
<th>2.0.3</th>
<th>2.1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td># of design elements (i.e. files)</td>
<td>2,840</td>
<td>2,841</td>
<td>2,841</td>
<td>3,136</td>
<td>3,296</td>
<td>3,304</td>
<td>3,308</td>
<td>3,444</td>
<td>3,502</td>
<td>3,568</td>
<td>3,609</td>
<td>3,632</td>
<td>3,736</td>
</tr>
<tr>
<td># of direct technical dependencies b/w files</td>
<td>17274</td>
<td>17301</td>
<td>17306</td>
<td>18838</td>
<td>20047</td>
<td>20061</td>
<td>20784</td>
<td>21067</td>
<td>21482</td>
<td>21774</td>
<td>22156</td>
<td>22914</td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>566</td>
<td>566</td>
<td>566</td>
<td>596</td>
<td>614</td>
<td>614</td>
<td>616</td>
<td>626</td>
<td>633</td>
<td>637</td>
<td>655</td>
<td>674</td>
<td></td>
</tr>
<tr>
<td># of indirect technical dependencies b/w files (Design Complexity)</td>
<td>241831</td>
<td>241656</td>
<td>241678</td>
<td>255015</td>
<td>264675</td>
<td>264676</td>
<td>264995</td>
<td>267733</td>
<td>282277</td>
<td>285365</td>
<td>288433</td>
<td>292772</td>
<td>301950</td>
</tr>
<tr>
<td>Mean</td>
<td>92.23</td>
<td>92.65</td>
<td>92.04</td>
<td>105.3</td>
<td>89.27</td>
<td>114.6</td>
<td>97.6</td>
<td>102.6</td>
<td>195.6</td>
<td>95.29</td>
<td>114.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>273.1</td>
<td>275.5</td>
<td>273.4</td>
<td>288.7</td>
<td>271.5</td>
<td>303.9</td>
<td>283.2</td>
<td>287.1</td>
<td>457.5</td>
<td>287.8</td>
<td>278.1</td>
<td>468.4</td>
<td></td>
</tr>
<tr>
<td>Min</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>1,832</td>
<td>1,832</td>
<td>1,831</td>
<td>1,844</td>
<td>1,871</td>
<td>1,871</td>
<td>1,884</td>
<td>1,939</td>
<td>1,947</td>
<td>1,955</td>
<td>1,976</td>
<td>2,013</td>
<td></td>
</tr>
<tr>
<td># of cross-module dependencies (Design Modularity)</td>
<td>3.61</td>
<td>3.419</td>
<td>3.461</td>
<td>3.253</td>
<td>3.15</td>
<td>3.004</td>
<td>3.154</td>
<td>3.057</td>
<td>2.227</td>
<td>2.342</td>
<td>2.85</td>
<td>3.106</td>
<td>3.239</td>
</tr>
<tr>
<td>Mean</td>
<td>17.5</td>
<td>16.02</td>
<td>16.09</td>
<td>16.14</td>
<td>15.06</td>
<td>14.57</td>
<td>15.67</td>
<td>15.94</td>
<td>10.5</td>
<td>10.98</td>
<td>13.76</td>
<td>15.28</td>
<td>16.05</td>
</tr>
<tr>
<td>Standard Deviation</td>
<td>521</td>
<td>438</td>
<td>440</td>
<td>479</td>
<td>464</td>
<td>384</td>
<td>460</td>
<td>520</td>
<td>311</td>
<td>339</td>
<td>410</td>
<td>479</td>
<td>476</td>
</tr>
<tr>
<td>Min</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Max</td>
<td>23</td>
<td>23</td>
<td>24</td>
<td>22</td>
<td>25</td>
<td>58</td>
<td>27</td>
<td>24</td>
<td>209</td>
<td>26</td>
<td>24</td>
<td>23</td>
<td>210</td>
</tr>
</tbody>
</table>

Table 1. Descriptive data of the analyzed software versions, measured at file level
<table>
<thead>
<tr>
<th>Variable</th>
<th>Measure</th>
<th># of obs.</th>
<th>Mean</th>
<th>Std. Dev.</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design Complexity</td>
<td>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</td>
<td>29705</td>
<td>238.597</td>
<td>375.822</td>
<td>1</td>
<td>2015</td>
</tr>
<tr>
<td>Design Modularity</td>
<td>(Reversely coded) Number of dependent elements lying outside of the focal element’s module</td>
<td>29705</td>
<td>1.512</td>
<td>14.21</td>
<td>0</td>
<td>521</td>
</tr>
<tr>
<td>Design Hierarchy</td>
<td>(Reversely coded) Number of elements reciprocally dependent on the focal design element</td>
<td>29705</td>
<td>3.823</td>
<td>24.81</td>
<td>0</td>
<td>210</td>
</tr>
<tr>
<td>Task Assignment</td>
<td>= 0 if the task on the focal element is performed by a single individual, = 1 if the task performed by a team of developers</td>
<td>11855</td>
<td>0.086</td>
<td>0.28</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Java Files</td>
<td>Identifies Java files from other file types.</td>
<td>29705</td>
<td>0.882</td>
<td>0.323</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Cyclomatic Complexity</td>
<td>Measurement of internal complexity of the focal element (file), measured by mining its code lines.</td>
<td>29705</td>
<td>5.942</td>
<td>8.991</td>
<td>0</td>
<td>127</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>Number of lines of code contained in the focal element (file)</td>
<td>29705</td>
<td>174.5</td>
<td>305.3</td>
<td>1</td>
<td>5,500</td>
</tr>
</tbody>
</table>
Table 3. Pairwise correlation results

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Design Complexity</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Design Hierarchy (Reversely coded)</td>
<td>-0.330</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Design Modularity (Reversely coded)</td>
<td>0.030</td>
<td>0.050</td>
<td>1.000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Java Files</td>
<td>0.030</td>
<td>0.030</td>
<td>0.040</td>
<td>1.000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Cyclomatic Complexity</td>
<td>0.000</td>
<td>0.030</td>
<td>0.000</td>
<td>0.040</td>
<td>1.000</td>
<td></td>
</tr>
<tr>
<td>6. Lines of Codes</td>
<td>0.000</td>
<td>0.040</td>
<td>0.020</td>
<td>-0.200</td>
<td>0.510</td>
<td>1.000</td>
</tr>
</tbody>
</table>

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, with random- and fixed-effects (H1 & H3)

<table>
<thead>
<tr>
<th>Dependent Variable: Design Complexity</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Modularity&lt;sub&gt;t-1&lt;/sub&gt;</td>
<td>3.592499***</td>
<td>3.460649***</td>
<td>4.353783**</td>
<td>4.381252**</td>
</tr>
<tr>
<td>[# of cross-modular dependencies]</td>
<td>(.5929136)</td>
<td>(.5900099)</td>
<td>(1.354099)</td>
<td>(1.335767)</td>
</tr>
<tr>
<td>Non-Hierarchy&lt;sub&gt;t-1&lt;/sub&gt;</td>
<td>7.24202***</td>
<td>7.170846***</td>
<td>2.1396</td>
<td>2.101103</td>
</tr>
<tr>
<td>[# of cyclic dependencies]</td>
<td>(.6866302)</td>
<td>(.6865654)</td>
<td>(1.23238)</td>
<td>(1.231275)</td>
</tr>
<tr>
<td>Non-Modularity&lt;sub&gt;t-1&lt;/sub&gt;&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-.0089444***</td>
<td>-.0091197***</td>
<td>-.0073909**</td>
<td>-.0074905**</td>
</tr>
<tr>
<td></td>
<td>(.002169)</td>
<td>(.0020873)</td>
<td>(.0024877)</td>
<td>(.0024679)</td>
</tr>
<tr>
<td>Non-Hierarchy&lt;sub&gt;t-1&lt;/sub&gt;&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-.0518911***</td>
<td>-.0519829***</td>
<td>-.0118143*</td>
<td>-.0117637*</td>
</tr>
<tr>
<td></td>
<td>(.0032729)</td>
<td>(.0032639)</td>
<td>(.0057723)</td>
<td>(.0057667)</td>
</tr>
<tr>
<td>Non-Modularity&lt;sub&gt;t-1&lt;/sub&gt; × Non-Hierarchy&lt;sub&gt;t-1&lt;/sub&gt;</td>
<td>.019196***</td>
<td>.019196***</td>
<td>.0050742*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(.0046088)</td>
<td>(.0046088)</td>
<td>(.0023436)</td>
<td></td>
</tr>
<tr>
<td>Fixed effects (of design elements)</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Constant</td>
<td>24.0801***</td>
<td>24.13905***</td>
<td>136.451***</td>
<td>136.314***</td>
</tr>
<tr>
<td></td>
<td>(1.501979)</td>
<td>(1.501998)</td>
<td>(3.566419)</td>
<td>(3.571464)</td>
</tr>
<tr>
<td>R-squared</td>
<td>0.1453</td>
<td>0.1460</td>
<td>0.4548398</td>
<td>0.4548846</td>
</tr>
<tr>
<td>Number of observations</td>
<td>25746</td>
<td>25746</td>
<td>25746</td>
<td>25746</td>
</tr>
</tbody>
</table>

Robust standard errors in parentheses, *** p<0.01, ** p<0.05, * p<0.1
Table 5. Pooled regression results of design complexity, with the interaction effect of Individual (H2) and Team (H4) Task Assignments

<table>
<thead>
<tr>
<th>Dependent Variable: Design Complexity</th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Modularity&lt;sub&gt;1&lt;/sub&gt;</td>
<td>30.806***</td>
<td>31.070***</td>
<td>30.698***</td>
<td>30.939***</td>
<td>30.620***</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[ # of cross-modular dependencies]</td>
<td>(1.18)</td>
<td>(1.19)</td>
<td>(1.11)</td>
<td>(1.18)</td>
<td>(1.10)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Hierarchy&lt;sub&gt;1&lt;/sub&gt;</td>
<td>5.216***</td>
<td></td>
<td>6.819***</td>
<td>6.120***</td>
<td>5.676***</td>
<td>5.555***</td>
<td></td>
</tr>
<tr>
<td>[ # of cyclic dependencies]</td>
<td>(1.31)</td>
<td></td>
<td>(1.39)</td>
<td>(1.31)</td>
<td>(1.38)</td>
<td>(1.27)</td>
<td></td>
</tr>
<tr>
<td>Task Assignment&lt;sup&gt;r&lt;/sup&gt;</td>
<td>4.829</td>
<td>17.440</td>
<td>14.751</td>
<td>13.973</td>
<td>12.602</td>
<td>22.001*</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(11.76)</td>
<td>(11.97)</td>
<td>(10.85)</td>
<td>(10.65)</td>
<td>(12.15)</td>
<td>(12.29)</td>
<td></td>
</tr>
<tr>
<td>Task Assignment&lt;sup&gt;r&lt;/sup&gt; X</td>
<td>-1.035</td>
<td>-2.356***</td>
<td>-1.069</td>
<td>-2.247**</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Modularity&lt;sub&gt;1&lt;/sub&gt;</td>
<td>(0.69)</td>
<td>(0.69)</td>
<td>(0.70)</td>
<td>(0.70)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Task Assignment&lt;sup&gt;r&lt;/sup&gt; X</td>
<td>-12.382**</td>
<td>-8.442**</td>
<td>-10.980***</td>
<td>-7.905***</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Hierarchy&lt;sub&gt;1&lt;/sub&gt;</td>
<td>(2.64)</td>
<td>(2.52)</td>
<td>(2.29)</td>
<td>(2.08)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Modularity&lt;sub&gt;1&lt;/sub&gt;&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-0.684***</td>
<td>-0.681***</td>
<td>-0.648***</td>
<td></td>
<td>-0.683***</td>
<td>-0.650***</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.04)</td>
<td>(0.04)</td>
<td>(0.04)</td>
<td></td>
<td>(0.04)</td>
<td>(0.04)</td>
<td></td>
</tr>
<tr>
<td>Non-Hierarchy&lt;sub&gt;1&lt;/sub&gt;&lt;sup&gt;2&lt;/sup&gt;</td>
<td>-0.023***</td>
<td>-0.030***</td>
<td>-0.026***</td>
<td>-0.025***</td>
<td>-0.024***</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(0.01)</td>
<td>(0.01)</td>
<td>(0.01)</td>
<td>(0.01)</td>
<td>(0.01)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Control Variables**

<table>
<thead>
<tr>
<th></th>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java Files&lt;sub&gt;1&lt;/sub&gt;</td>
<td>199.480***</td>
<td>189.910***</td>
<td>184.894***</td>
<td>341.161***</td>
<td>328.355***</td>
<td>200.654***</td>
<td>198.689***</td>
</tr>
<tr>
<td></td>
<td>(5.71)</td>
<td>(5.36)</td>
<td>(7.66)</td>
<td>(5.35)</td>
<td>(8.16)</td>
<td>(5.80)</td>
<td>(7.89)</td>
</tr>
<tr>
<td>Cyclomatic Complexity&lt;sub&gt;1&lt;/sub&gt;</td>
<td>1.017***</td>
<td>1.090***</td>
<td>0.781***</td>
<td>3.154***</td>
<td>2.786***</td>
<td>1.049***</td>
<td>0.751***</td>
</tr>
<tr>
<td></td>
<td>(0.15)</td>
<td>(0.16)</td>
<td>(0.16)</td>
<td>(0.16)</td>
<td>(0.16)</td>
<td>(0.16)</td>
<td>(0.16)</td>
</tr>
<tr>
<td>Lines of Code&lt;sub&gt;1&lt;/sub&gt;</td>
<td>0.190 (0.50)</td>
<td>-0.248 (0.54)</td>
<td>-0.947* (0.57)</td>
<td>3.318*** (0.64)</td>
<td>3.218** (0.67)</td>
<td>0.017 (0.54)</td>
<td>-0.870 (0.57)</td>
</tr>
<tr>
<td>Constant</td>
<td>-0.045 (3.61)</td>
<td>11.411*** (2.65)</td>
<td>-11.972 (12.43)</td>
<td>-40.351*** (4.17)</td>
<td>-58.017* (25.63)</td>
<td>-2.034 (3.81)</td>
<td>-13.188 (13.23)</td>
</tr>
</tbody>
</table>

Version’s Fixed Effect Y Y Y Y
Developers’ Fixed Effects Y Y Y Y
R-squared 0.388 0.384 0.479 0.296 0.394 0.388 0.483
Number of obs. 6675.000 6675.000 6675.000 6675.000 6675.000 6675.000 6675.000

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


