

# Singapore Management University Institutional Knowledge at Singapore Management University

---

Research Collection School Of Information Systems

School of Information Systems

---


5-2007

## Modeling Architectural Strategy Using Design Structure Networks

C. Jason WOODARD

Singapore Management University, [jason.woodard@olin.edu](mailto:jason.woodard@olin.edu)

Follow this and additional works at: [https://ink.library.smu.edu.sg/sis\\_research](https://ink.library.smu.edu.sg/sis_research)

 Part of the [Computer Sciences Commons](#), and the [Management Information Systems Commons](#)

---

### Citation

WOODARD, C. Jason. Modeling Architectural Strategy Using Design Structure Networks. (2007). *2nd International Conference on Design Science Research in Information Systems and Technology (DESRIST 2007)*. Research Collection School Of Information Systems. **Available at:** [https://ink.library.smu.edu.sg/sis\\_research/736](https://ink.library.smu.edu.sg/sis_research/736)

This Conference Paper is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email [libIR@smu.edu.sg](mailto:libIR@smu.edu.sg).

# Modeling Architectural Strategy Using Design Structure Networks

C. Jason Woodard\*

School of Information Systems  
Singapore Management University  
jwoodard@smu.edu.sg

21 April 2007

## Abstract

System architects face the formidable task of purposefully shaping an evolving space of complex designs. Their task is further complicated when they lack full control of the design process, and therefore must anticipate the behavior of other stakeholders, including the designers of component products and competing systems. This paper presents a conceptual tool called a design structure network (DSN) to help architects and design scientists reason effectively about these situations. A DSN is a graphical representation of a system's design space. DSNs improve on existing representation schemes by providing a compact and intuitive way to express design options—the ability to replace all or part of one design with another. Design options, in turn, are the building blocks of architectural strategy—the practice of designing systems with an awareness that the fortunes of other designers are intertwined with one's own. I illustrate the informal use of design structure networks with an example based on Apple's decision to adopt the Intel processor architecture. I also show how DSNs can serve as a formal foundation for economic models of architectural strategy, which I call system design games.

---

\*Presented at the Second International Conference on Design Science Research in Information Systems and Technology (DESRIST 2007) in Pasadena CA, 13–15 May 2007. This paper is based on the second chapter of my dissertation. It has benefited greatly from the input of my committee, especially Carliss Baldwin and David Parkes, and from the feedback of an anonymous referee. All errors are my own.

# 1 Introduction

System architects have a hard job. They are expected to meet the demanding, conflicting, and often unstated requirements of numerous stakeholders. They are frequently accused of meddling in implementation details, but held responsible when components fail to work together. They become entangled in organizational politics because the structure of a system is entangled with the division of labor needed to build it. With all this to worry about, it is easy to forget that other architects may be conspiring against them.

A healthy dose of paranoia (cf. Grove 1996) is warranted by the fact that most large systems have many designers, each of whom controls a different part of the design and pursues a distinct set of objectives. This can be true even in projects sponsored by a single firm, such as the IBM System/360 family of mainframe computers, where the goal of compatibility across the family put designers of the high-end machines at odds with their colleagues at the low end of the range (Pugh et al. 1991; Baldwin and Clark 2000, ch. 7). It is emphatically the case in systems whose components are supplied by competing firms, where contention over the control of architectural standards is pervasive and intense (Langlois and Robertson 1992; Morris and Ferguson 1993; Katz and Shapiro 1994). As advances in software engineering make possible “ultra-large-scale” systems with over a billion lines of code (Northrop et al. 2006), the dispersion of managerial control and consequent rise of incentive conflicts can only be expected to increase.

*Architectural strategy* is the practice of designing systems with an awareness that the fortunes of other designers are intertwined with one’s own. The design process takes place in an evolving space of possible and actual designs for the system and its components. This space may be large, complex, and difficult to observe, posing a challenge to understanding the interrelated choices available to designers. In this paper, I present a conceptual tool called a *design structure network* (DSN) to help architects and design scientists reason effectively about these situations.

A DSN is a graphical representation of a system’s design space. DSNs improve on existing representation schemes by providing a compact and intuitive way to express *design options*—the ability to replace all or part of one design with another. Design options, in turn, are the building blocks of architectural strategy. For designers who “see and seek value” (Baldwin and Clark 2000), the design problem boils down to evaluating the costs and benefits of each available option—taking into account its likely effects on other stakeholders and their anticipated responses, as well as its foreseeable impact on future decisions—and exercising the options that are expected to yield the greatest net value for the designer.

## 1.1 Related work

Techniques for representing system architectures have been developed in a variety of domains. Software-intensive systems have attracted particular attention because of the difficulty in visualizing the complex interactions among their many parts. Software engineers broadly concur that no single representation can capture all of the important

entities and relationships in a software system, so it is generally useful to create several different graphical “views” of the system (Krutchen 1995; Bass et al. 1998; Clements et al. 2002). An active research community has provided increasingly rigorous underpinnings for these views, taking software architecture from an “emerging discipline” (Garlan and Shaw 1996) to its “golden age” (Shaw and Clements 2006) in the short span of a decade.

The *design structure matrix* (DSM) technique builds on a complementary base of research that spans engineering disciplines (Steward 1981; Eppinger et al. 1994; Sharman and Yassine 2004). In a DSM, matrix elements represent tasks or decisions, and a nonzero entry in cell  $(i, j)$  indicates that  $i$  depends on  $j$ . A DSM thus contains the same information as a directed graph with an arrow to each task or decision from each of its dependents. As with software architecture diagrams, DSMs can be used at different levels of analysis (e.g., for components, systems, or even systems of systems) to describe different kinds of dependencies (e.g., in design, production, or use). DSMs have been used to study designs in settings as diverse as building construction, semiconductor manufacturing, and the automotive and aerospace industries (Browning 2001).

A design structure network goes beyond the description of a single design to represent a set of possible designs—that is, the *design space* of an artifact or system. To enumerate these designs exhaustively is generally a hopeless task, since their number may be vast. However, it is often fruitful to examine the relationships among related subsets of designs. Consider the design of a typical information system, in which several key components (e.g., database software, web application server, and messaging middleware) are supplied by commercial vendors. The choice of these components may profoundly affect the system’s future evolution as a result of vendor “lock-in” (Arthur 1989; Farrell and Saloner 1992; Shapiro and Varian 1999). Examining the regions of the design space that differ in the component products chosen may be both feasible and valuable for the system architect. Of course, the same applies to the vendors, whose interest lies in creating products that come out favorably in this kind of analysis.

DSMs lack a natural way to represent choices among component designs, which Baldwin and Clark (2000) call design options.<sup>1</sup> To analyze systems with design options, we need to be able to represent component designs that “overlap” in the sense that they are at least partially interchangeable. Design structure networks provide this capability. While this is true of some architectural specification languages in principle—for example, the Unified Modeling Language (Fowler 2003) is expressive enough to define DSNs using its extension mechanism—option-based views are not yet part of the standard toolkit of practicing software architects. The only comparable approach of which I am aware is that of Cai and Sullivan (2005a,b) who develop a complementary formalism based on *augmented constraint networks* (ACNs). Their approach yields a set of logical relationships that can be analyzed using automated constraint solvers like Alloy (Jackson 2002), whereas DSNs are designed to highlight the most economically significant relationships among components for the purposes of managerial decision-making and economic modeling.

---

<sup>1</sup>Baldwin and Clark, in turn, build on the vast literature concerning financial and real options (Merton 1998; Amram and Kulatilaka 1999).

## 1.2 Contributions to design science

This paper contributes to the growing body of research on the science of design by proposing a new way to represent system design spaces, with a focus on modular designs such as those typical of information systems and other IT artifacts. As noted by Hevner et al. (2004), Herbert Simon stressed the importance of appropriate representations in his well-known characterization of problem solving: “Solving a problem simply means representing it so as to make the solution transparent” (1996, p. 132).

While I have not yet evaluated the utility of DSN representations in solving real-world problems of strategic system design, section 2 provides an example of the kind of problems they are designed to shed light on. Section 3 lays out the DSN formalism in mathematical notation to aid verification of its internal consistency. Section 4 shows that DSNs capture familiar properties of systems, such as information hiding and compatibility, in an intuitive way. Sections 5 and 6 discuss the application of DSNs to analytical and computational modeling, respectively, using the principles of game theory and complex adaptive systems. Section 7 concludes.

## 2 Architectural strategy at Apple

Apple Computer was famous for its proprietary Macintosh computer design and dogmatic rejection of technologies that would put its products into more direct competition with makers of “IBM-compatible” personal computers (PCs). In June 2005, however, the company announced that it would switch from the PowerPC processor architecture it had championed with IBM and Motorola to the Intel architecture favored by the rest of the PC industry.<sup>2</sup> In April 2006, the company committed the even greater heresy (in the eyes of Mac loyalists) by releasing a piece of software called Boot Camp that enabled Intel-based Macs to run Microsoft Windows.<sup>3</sup>

In this section I want to illustrate these “design moves” using design structure networks. The purpose of this exercise is not to probe deeply into the rationale behind Apple’s decisions or their consequences for the PC industry—this would ask too much of a toy example and take us too far afield—but simply to show that DSNs provide a way to start discussing these issues with greater precision than one might achieve through verbal descriptions or ad hoc diagrams alone. Consider the two diagrams in figures 1 and 2. Sections 3 and 4 provide a formal basis for them, but with a few hints we can interpret them informally using language familiar to industry observers.

Figure 1 illustrates the situation of Apple and its competitors before the Apple–Intel announcement. The figure shows two “stacks,” each consisting of a processor architecture, a family of computer systems, and an operating system (OS). The arrows indicate technological dependencies: each OS depends on a computer system to operate correctly, and each computer system in turn depends on a processor. The fact that the arrow emanating from the oval labeled “Microsoft Windows” attaches to

---

<sup>2</sup>Apple Computer, “Apple to Use Intel Microprocessors Beginning in 2006,” June 6, 2005, <http://www.apple.com/pr/library/2005/jun/06intel.html>.

<sup>3</sup>Apple Computer, “Apple Introduces Boot Camp,” April 5, 2006, <http://www.apple.com/pr/library/2006/apr/05bootcamp.html>.

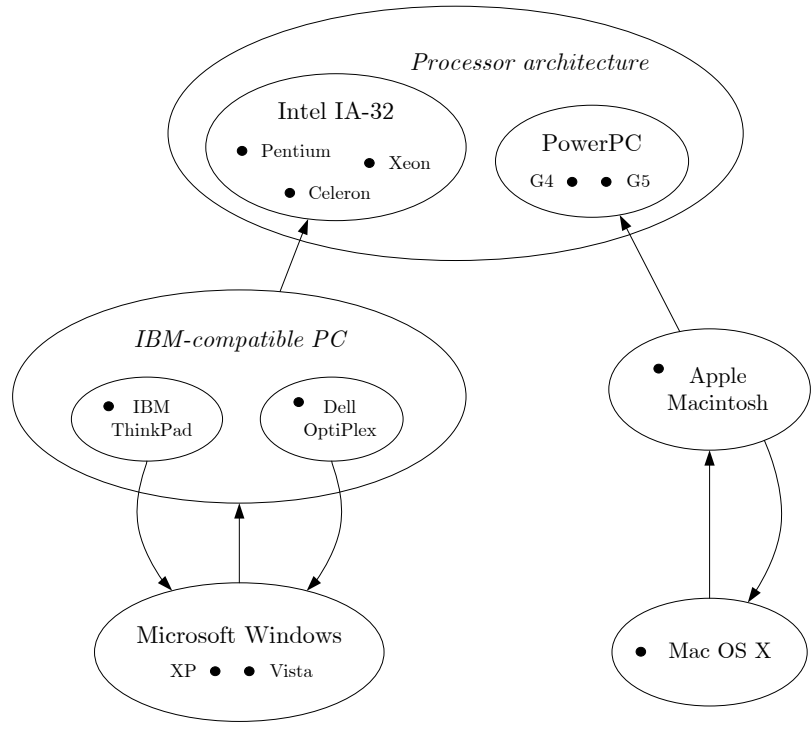


Figure 1: IBM-compatible PCs and the PowerPC-based Macintosh.

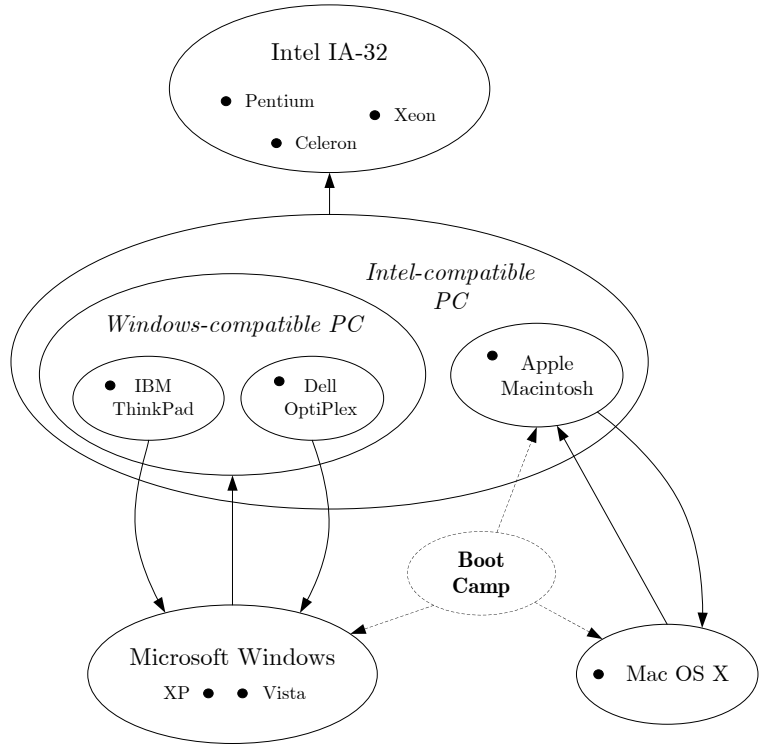


Figure 2: The new Intel-based Macintosh and Apple's Boot Camp software.

the one labeled “IBM-compatible PC” indicates that Windows depends on an abstract design, of which both IBM (now Lenovo) ThinkPads and Dell-branded computers are instances.<sup>4</sup> ThinkPads and Dell PCs are thus “clones” of each other with respect to Windows. Moreover, there is a dependency arrow from each brand of PC back to Windows. These indicate that each vendor’s products include features, such as hardware drivers and accessory software, that are specially designed to be used with Windows.<sup>5</sup> The Macintosh family follows a similar pattern, yielding a circular chain of dependence between the hardware and the Mac OS.

Figure 2 shows the situation after the Apple–Intel announcement, with the subsequent release of the Boot Camp software indicated using dashed lines. The Intel-based Macintosh is now a sibling of the IBM and Dell computers within the product category labeled “Intel-compatible PC.” However, the Mac and its OS remain oblivious to Windows, and vice versa. Boot Camp provides the missing links: a set of drivers that allow Windows to recognize the Mac’s unique hardware components, an update for the Mac’s firmware to emulate the PC BIOS on which Windows relies for low-level system tasks, and a small utility program called a boot loader to allow the user to select OS X or Windows when the system is started.

The two DSNs thus convey important information about personal computer designs, including which components we would expect to be economic complements (e.g., PCs, processors, and operating systems) and which should be substitutes (e.g., ThinkPads and Dell laptops). The DSNs also convey information about designs that are not shown explicitly in either figure; this is the sense in which a DSN represents not merely a design but a design space. For example, consider adding a dot labeled “AMD Athlon” to the “Intel IA-32” oval. We can infer that the product represented by this dot—which was, in fact, released in 1999—would work with IBM-compatible (a.k.a. “Wintel”) PCs and compete with Intel’s 32-bit processors. We can also imagine alternative component designs with different dependencies, such as an IBM ThinkPad that runs on the PowerPC processor (which also existed at one time). This would be the result of a design move that changes the design space, which Baldwin and Clark (2000) call the application of a *modular operator*.

While these two simple DSNs do not provide enough information to predict how the Apple–Intel–Microsoft story will unfold, they suggest several possibilities:

- Apple could make it possible to run the Mac OS on other vendors’ hardware. This would entail a similar set of technologies to Boot Camp, with different device drivers and a way of adapting the OS to use the standard PC BIOS instead of adapting Windows to use the Mac firmware.

---

<sup>4</sup>Historically, this abstract design was the de facto standard established by IBM. With the rise of Windows in the 1990s, Microsoft and Intel displaced IBM from its position of power to set the design rules for personal computers. Increasingly, PC vendors worked to ensure compatibility with the Windows operating system and Intel chipsets rather than the other way around. (See, e.g., [http://en.wikipedia.org/wiki/IBM\\_PC\\_compatible](http://en.wikipedia.org/wiki/IBM_PC_compatible).)

<sup>5</sup>The fact that these vendors make use of design information specific to Windows is made evident by their participation in Microsoft’s “Designed for Windows” certification program (<http://www.microsoft.com/winlogo/>).

- Microsoft could “embrace and extend” the Macintosh hardware platform in an attempt to marginalize OS X. Of course, doing this successfully would require some degree of cooperation (or at least the absence of opposition) from Apple, which controls the Mac hardware specifications.
- Intel could provide new opportunities for differentiation in the PC market, by Apple or others, by creating a “premium” processor architecture that is incompatible with IA-32. (The result of this actual decision was the less-than-successful Itanium product line.)

The DSN diagrams thus serve to bring into focus not just the existing positions of the various firms in Apple’s competitive landscape, but also some of the design moves available to them.

Analyzing these moves requires us to consider the way each would affect the economic value created by the PC industry and the relative ability of each firm to capture a share of this value. This kind of analysis is facilitated by the economic modeling techniques introduced in sections 5 and 6. The main goal of the remainder of the paper, however, is narrower in scope—namely to solidify the formal and conceptual foundations of the DSN representation scheme.

### 3 Formal definitions

A design structure network is a mathematical description of a system. A *system* is a collection of *components*: technologically and economically discrete parts that are designed to work together.<sup>6</sup> Each component is associated with a *design space* defined by a set of *design variables*. If  $i$  is a component with  $d_i$  design variables, let  $X^i = X_1^i \times \cdots \times X_{d_i}^i$  denote its design space, where  $X_k^i$  is a set of possible values called the *domain* of variable  $k$ . Let  $M$  be the set of components in the system of interest, and  $X = \times_{i \in M} X^i$  denote the *system design space*.<sup>7</sup>

A design is associated with a region in a design space called a *type*. Designs can be complete or incomplete. A *product* is a complete design for a component, represented as a point in the component’s design space. We can write a typical product  $x^i \in X^i$  as a tuple  $(x_1^i, \dots, x_{d_i}^i)$  indicating the chosen value of each variable. For the special case in which all variables take on binary values, we can write  $x^i$  as a sequence of 1s and 0s, commonly known as a “bit string.” In fact, any design can be represented as a bit string by converting each of its values from its natural domain into a binary equivalent (e.g.,  $5 \rightarrow 0101$ ) and concatenating the results. This idea is familiar to software engineers; a similar process takes place every time a program saves a file to disk.

---

<sup>6</sup>This definition is intended to mirror the use of the terms “system” and “component” in ordinary language, where they can refer to abstract designs (e.g., “the global transportation system”) as well as concrete artifacts (e.g., “my new stereo system”). Some ambiguity is inevitable as a result, although as in ordinary language the exact meaning is usually clear from context.

<sup>7</sup>Many design theorists have employed the idea of a design space defined by a set of orthogonal design variables. My setup is similar to that of Cai and Sullivan (2005a,b). I believe the connection between DSNs and their constraint networks can be made precise, but have not yet verified this.



In an incomplete design, one or more design variables are not completely specified. A type thus refers to an ensemble of related designs. This usage is consistent with both standard English and computer science. According to the Merriam-Webster Online Dictionary, a type is “a particular kind, class, or group” (def. 4d) as well as a bundle of “qualities common to a number of individuals that distinguish them as an identifiable class” (def. 4a). Computer scientists use the term in the latter sense, as a range of values that a variable can assume during execution (Cardelli 1997; Pierce 2002). In the context of designs, a type defines a set of design options.

Any subset of a component’s design space is a valid type. Even in simple systems, though, the number of valid types far exceeds the ability of a designer to reason about them all. (For a component with eight binary design variables there are  $2^{2^8} \approx 10^{77}$  distinct types, counting every possible subset of the  $2^8 = 256$  distinct 8-bit component designs—comparable to the estimated number of atoms in the universe.) We therefore restrict attention to a subset of types,  $T^i \subseteq 2^{X^i}$ , where  $2^{X^i}$  is the power set of component  $i$ ’s design space. This subset, called the *focal types* for component  $i$ , is specified by the modeler according to the situation of interest.<sup>8</sup> We can then write  $j \in T^i$  to denote a particular type, where  $j \subseteq X^i$ . Let  $T = \bigcup_{i \in M} T^i$  be the set of focal types for the system.

As a set of possibly overlapping sets,  $T$  is akin to a collection of Venn diagrams. Their boundaries are summarized compactly by the graph  $(T, E)$ , where  $E \subseteq T \times T$  is a set of edges such that  $(j', j) \in E$  if and only if  $j$  strictly covers  $j'$ , i.e.,  $j' \subset j$  and there is no  $j'' \in T$  such that  $j' \subset j'' \subset j$ . We call  $(T, E)$  the *inheritance graph* for the system, and say that  $j'$  *extends*  $j$  if  $(j', j) \in E$ . For any pair of types  $j$  and  $j'$ , if  $j' \subseteq j$ , then  $j'$  is a *subtype* of  $j$  and  $j$  is a *supertype* of  $j'$ .

As in a DSM, we need a way to record the dependencies among types. Unlike inheritance relationships, dependencies generally cannot be inferred from the structure of a system’s design space, so we treat them as inputs to the model, like the focal types. Let  $D \subseteq T \times T$  be a set of *direct dependencies*, and call  $(T, D)$  the *dependence graph* for the system. Although it is perfectly valid for one type of component to depend on another type of the same component (e.g., as left and right stereo speakers might depend on each other), dependencies are more commonly used to express interactions across components (e.g., between speakers and tuners).

From the direct dependencies and the inheritance graph we also define a set of *induced dependencies*,  $\mathcal{D} \subseteq T \times T$ , where for all  $(i, j) \in \mathcal{D}$  there is an edge  $(i', j') \in D$  for each subtype of  $i$  and each supertype of  $j$ , denoted  $i'$  and  $j'$  respectively.  $\mathcal{D}$  specifies how dependencies propagate through the type hierarchy. If  $(i, j) \in \mathcal{D}$ , we say that  $i$  *depends on*  $j$ . If  $(i, j) \notin \mathcal{D}$  and  $(j, i) \notin \mathcal{D}$ , then  $i$  and  $j$  are *independent*.

Finally, we call  $G_X = (T, D, E)$  a *design structure network* for the design space  $X$ . Note that DSNs are strictly more expressive than design structure matrices: any DSM can be represented as a DSN by assigning each design variable to a separate component with a single type, thus letting  $E$  be empty and  $\mathcal{D} = D$ .

---

<sup>8</sup>A component’s entire design space may be a focal type (in which case  $X^i \in T^i$ ), though it need not be. A region containing a single product, e.g.,  $x^i \in X^i$ , may also be a focal type (in which case  $\{x^i\} \in T^i$ ). These possibilities justify the idiom of treating components and products as types where unambiguous.

## 4 Key properties of DSN representations

DSNs capture familiar properties of systems in an intuitive way, as I show in this section using an example of a system related to personal transportation. Figure 3 shows a DSN for part of this system. Each oval denotes a type, including three corresponding to top-level components (roads, cars, and fuel) and eight to nested ones, such as paved roads and unleaded gasoline. Some types are unnamed; their meaning will become clear from context. The solid dots show nine products. Nesting the ovals indicates inheritance: if  $i$ 's oval is directly enclosed by  $j$ 's, then  $i$  extends  $j$ . An arrow from  $i$ 's oval to  $j$ 's indicates that type  $i$  depends on  $j$ . Figure 4 shows the equivalent dependence and inheritance graphs separately using traditional notation. The numbered nodes refer to types labeled in the original figure.<sup>9</sup>

### 4.1 Dependence and inheritance

How should we interpret the relationships represented by this DSN? Consider inheritance first. From figure 3, we can infer that a paved road shares all the features of a generic road (e.g., it is long and narrow) with additional distinguishing characteristics (e.g., a hard, smooth surface). Similarly, all fuels supply energy, but in different forms; unleaded gasoline is a liquid at standard temperature and pressure, unlike compressed natural gas (CNG). The nested ovals thus convey the “narrowing” of each component’s design space. The product labeled Mobil unleaded gasoline reflects a specific set of design choices that determine its chemical composition and performance characteristics. Shell’s product may differ in some of these choices (e.g., additives to reduce emissions or engine deposits), but only insofar as it remains unleaded gasoline and not some other type of fuel.<sup>10</sup>

Dependence relationships can also be read from the figure. The arrow to unleaded gasoline from the type labeled 7, which includes the Ford Explorer and the Toyota Camry, indicates that in designing these cars, engineers anticipated the use of unleaded gasoline as a fuel, but made their designs independent of the gasoline brand. (If we wanted to name this type, we could call it “unleaded gasoline-burning car.”) While the Honda Civic GX runs on compressed natural gas instead of gasoline, the Civic and the Camry both require paved roads.<sup>11</sup> The Ford Explorer, in contrast, is

---

<sup>9</sup>Using this numbering scheme, the complete DSN for the labeled types would be given by  $T = \{1, \dots, 11\}$ ,  $D = \{(7, 4), (8, 2), (9, 1), (11, 5)\}$ , and  $E = \{(2, 1), (4, 3), (5, 3), (7, 6), (8, 6), (9, 7), (10, 7), (10, 8), (11, 8)\}$ . Here natural numbers are used as identifiers of types, which are defined as sets in section 3. This is simply a labeling convention; we could just as easily use letters or any other symbols. Since the inheritance relationships for the products are straightforward and they do not participate in any direct dependencies, I omit them from figure 4 and the enumeration here.

<sup>10</sup>Gasoline is in fact a complex product, varying not only in octane rating—an obvious omission from the figure—but along dozens of other dimensions as well. Even identically labeled products are formulated differently by region and season to account for the effects of altitude and temperature on combustion, and to comply with local regulations. See <http://www.faqs.org/faqs/autos/gasoline-faq/>.

<sup>11</sup>The Camry thus inherits attributes from types 7 and 8 through its parent type, labeled 10. This kind of “multiple inheritance” results in an inheritance graph that is a directed acyclic graph (DAG), rather than a tree, because some nodes have more than one parent.

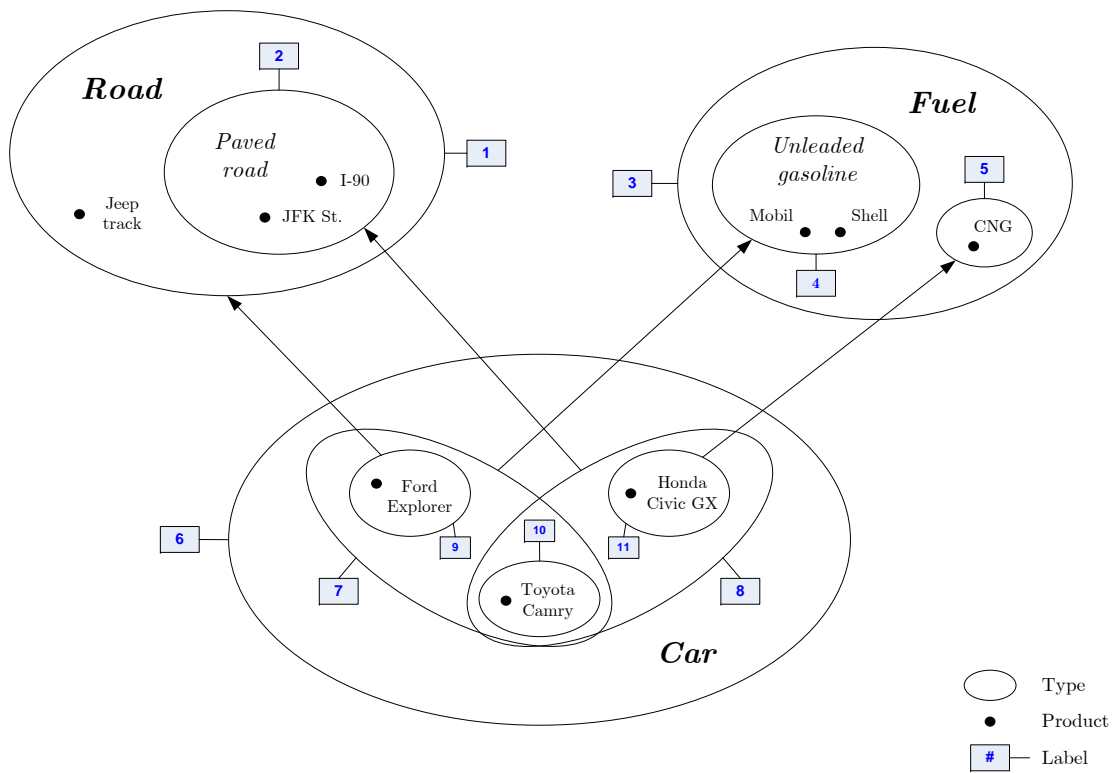


Figure 3: An automotive design structure network.

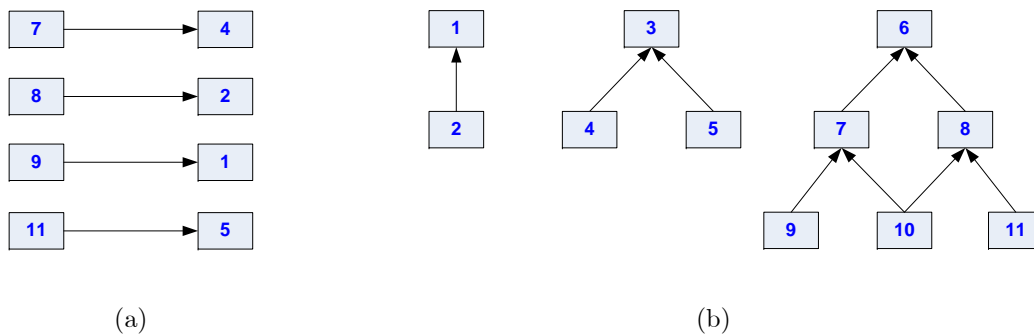


Figure 4: (a) Dependence graph, and (b) inheritance graph for the automotive DSN.

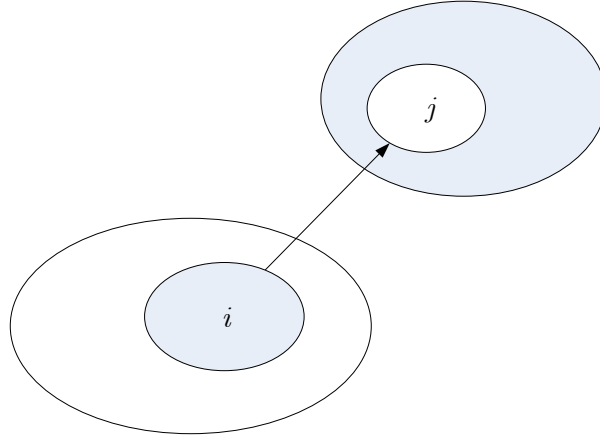


Figure 5: Induced dependence: subtypes of  $i$  depend on supertypes of  $j$ .

designed for a wider range of terrain including non-paved roads such as jeep tracks.

So what exactly is a dependency? The many answers given in the literature are similar in spirit, though they vary in emphasis and implications. Pimmler and Eppinger (1994) adopt the view that dependencies arise from interactions among design elements. They classify these interactions into four kinds: spatial proximity, energy exchange, information exchange, and materials exchange. Jackson (2003) characterizes dependencies as assumptions, stipulating that “a module  $A$  depends on a module  $B$  if the designer or implementer of  $A$  makes an assumption about the environment in which  $A$  is to be inserted that is justified in the constructed system by the presence of  $B$ .” MacCormack et al. (2006) and Sangal et al. (2005) take a pragmatic approach suited to the analysis of large software projects, inferring dependencies from function calls between source files and syntactic references between classes, respectively.

The DSN formalism does not attempt to define the concept precisely. DSNs share this feature with DSMs, to the possible frustration of researchers who find that dependence semantics are often poorly specified in the DSM literature (Cai and Sullivan 2005a).<sup>12</sup> However, DSNs are semantically richer than DSMs because their dependencies are defined with respect to types rather than individual design variables. If type  $i$  depends on  $j$ , the concept of induced dependence tells us that every subtype of  $i$  also depends on every supertype of  $j$ , as illustrated in figure 5. To see why this additional structure is natural, we need to reexamine the formalism through the lens of information hiding.

## 4.2 Interfaces and information

Hiding design information behind an interface is a common engineering practice. In a modular system, information hiding can reduce the cost of design changes, facilitate

---

<sup>12</sup>I am guilty of the same ambiguity in describing the automotive example. Phrases like “anticipated the use of,” “runs on,” and “require” connote different kinds of dependence. Value functions, introduced in section 5, provide a way to distinguish them.

independent development efforts, and make the system more comprehensible (Parnas 1972). These benefits are especially important in multi-product systems where components produced by different firms must interoperate. Parnas observes that a good modularization reduces the complexity of the interfaces exposed by each module, thus reducing the amount of coordination required across module boundaries.

Baldwin and Clark, following Parnas, characterize interfaces as “visible information” used in design (2000, p. 73). More concretely, they view an interface as a “treaty” among designers that partitions a module’s design variables into visible and hidden subsets. An interface specification fixes the values of the visible variables or restricts them to a certain range; a valid implementation of the interface must respect these choices. Hidden variables remain free to be chosen by implementers; modules that rely on an interface should function correctly under any values in these variables’ domains.

It is useful to interpret a type as a bundle of design information that defines an interface. Recall that a type specifies a set of possible designs for a component. In many cases (arguably the most interesting ones), the set corresponds to a partial design, which fixes or restricts some variables while leaving others free to be determined by subtypes.<sup>13</sup> The information associated with such a type can be expressed concisely as a *schema* (Holland 1992), using Holland’s modified bit-string notation with a “don’t care” symbol to indicate the free variables. For example,  $01\square11\square$  denotes a six-bit type whose third and sixth variables are free. This string represents the four-element subset  $\{010110, 010111, 011110, 011111\}$ .

As the set of possible designs for a given component shrinks with each nested subtype, the information content of each interface grows. Intuitively this is true because each subtype requires a more specific description than its supertype. For example, a generic car can be described in a few words (four wheels, internal combustion engine, etc.), while describing a Toyota Camry LE takes enough information to distinguish it from all other makes and models. The dual relationship between types and interfaces can be made precise using Shannon’s (1948) conception of information as entropy (Woodard 2006, ch. 2); space constraints preclude illustrating this here.

### 4.3 Congruence and compatibility

Returning to the graphical notation of figure 3, we can now say more about what it means for each oval to denote a type. An oval delimits both a boundary and an interior. The boundary represents the visible information associated with a type (its interface), while the interior represents the set of designs that conform to it. A dependency arrow thus indicates that one set of designs depends on information exposed by another. We can interpret the induced dependence rule as a simple consequence of the way design information propagates through type inheritance. Since subtypes inherit information from their parents, they need to know everything their parents do—so if  $i$  depends on  $j$ , any subtype of  $i$  will also depend on  $j$ . Conversely, information that needs to be

---

<sup>13</sup>As an example of a type that does not, in general, correspond to a partial design, consider one that consists of products chosen uniformly at random from a design space.

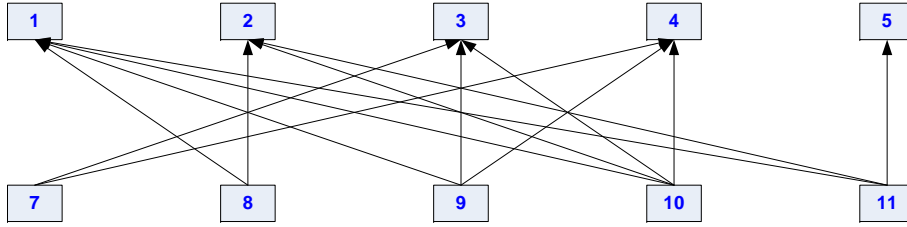


Figure 6: Induced dependencies for the automotive DSN.

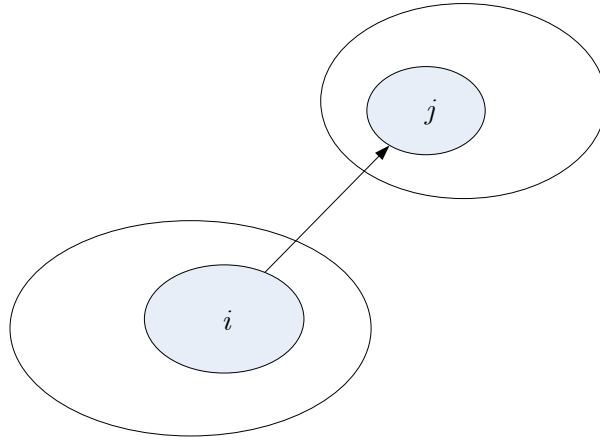


Figure 7: Congruence: subtypes of  $j$  are congruent with respect to subtypes of  $i$ .

known may have been inherited from a supertype—so we say that  $i$  and its subtypes also depend on the subtypes of  $j$ .

Figure 6 shows the full set of induced dependencies for the automotive DSN. We can see explicitly that any product of type 7, including those of types 9 and 10, depends on properties of both fuel in general (type 3) and unleaded gasoline in particular (type 4). Similarly, products of type 8, and by extension those of types 10 and 11, depend on roads in general (type 1) and paved roads in particular (type 2). These are exactly the relationships we inferred from the original figure; figure 6 contains no additional information. However, the new figure makes it easier to see which types do *not* depend on each other. We can use these independence relationships to draw inferences about a certain kind of compatibility between components.

Consider again the two brands of unleaded gasoline. The fact that the two gasoline-burning cars are independent of the two brands of gasoline means that either brand can be used with either car. In other words, since neither car incorporates brand-specific information into its design, the cars are oblivious to substitution among gasoline brands. More generally, we will say that types  $j'$  and  $j''$  are *congruent* with respect to  $i$  and its subtypes if  $i$  depends on a common supertype of  $j'$  and  $j''$  but is independent of both. Figure 7 illustrates this relationship graphically.

A value function, introduced in 5, may be needed to support more subtle inferences about compatibility as it affects product performance or economic value. Suppose the Mobil gasoline in the example has an octane rating of 93, while Shell's has a rating

of 87. Both cars might run with either brand, but they might run better with premium fuel. In other words, the fuels might be perfectly interchangeable in a technical sense, but not perfect economic substitutes. It should not be surprising that we need more than a structural description of the system to capture this distinction. More surprising, perhaps, is that we can get so far using only the DSN formalism.

#### 4.4 Abstraction to reduced form

Although I did not specify a design space for the automotive system example, its inheritance graph presumes the existence of one with the nesting structure illustrated in figure 3. Similarly, the discussion of the information content of types and interfaces appealed to the existence of a bit-string representation of product designs. However, actually enumerating the design variables and their domains may be a laborious undertaking, even for simple systems like the coffee cup example of Baldwin and Clark (2000, pp. 39–42). Fortunately, this exercise is unnecessary to work with a DSN representation.

Consider the design space of a coffee cup that is defined by a set of variables describing the cup’s shape, size, construction, and decoration. To refer to the set of orange ceramic coffee cups, we could write down a string with  $\square$ ’s (free variables) in all positions except those that correspond to color and material. But if we are only interested in the fact that certain mugs sold at Princeton are members of the set while those sold at Harvard are not, we can record this information without reference to the variables or their domains.

Many problems related to strategic system design are most naturally framed at a similar level of abstraction. To accommodate this, DSNs can be specified without an explicit design space. In this *reduced form*, a DSN is a tuple  $G = (T, D, E)$  in which the set of types  $T$  contains primitive elements rather than sets, and the inheritance graph  $E$ , like the dependence graph  $D$ , is given by the modeler rather than derived from the structure of the type space. A reduced-form DSN is sufficient to represent a design space in the notation of figure 3, and vice versa.<sup>14</sup> By freeing ourselves from the microstructure of design spaces, we can focus on the structural changes that occur when modular designs are reconfigured, along with the strategic implications of those changes.

---

<sup>14</sup>We have already seen an example of a mapping from a diagram to set notation: the enumeration given in footnote 9 is, in fact, a reduced form for the automotive DSN. Now the numbers are purely identifiers of types; we no longer need to think of them as sets of designs.

## 5 Beyond snapshots: System design games

Design structure networks provide graphical snapshots that may reveal opportunities or tensions among designers, shedding light on past events or yielding insight into future decisions. Unlike other representation schemes used in engineering design, DSNs illustrate economic relationships like complementarity and substitutability among a set of products or system components. A typical next step in analyzing these relationships is to study the incentives of the designers, which are not necessarily aligned with each other or those of their stakeholders. In particular, we often want to examine the forces for and against the various design moves available to system designers. These forces are naturally described using the language of game theory.

In an expanded version of this paper (Woodard 2006, ch. 2), I introduce the concept of a *system design game* (SDG) as an organizing framework for modeling the evolution of designs through the strategic decisions of designers. An SDG consists of three parts, corresponding to the core evolutionary processes of variation and selection plus a fitness or value function that serves as a proxy for the environment. The engine of variation is the use of *modular operators* by value-seeking designers. Baldwin and Clark, following John Holland’s (1992) seminal work on complex adaptive systems, identify six operators on modular designs—splitting, substitution, augmentation, exclusion, inversion, and porting—to which they attribute the historical explosion in the diversity and complexity of computer systems. I formalize a modular operator as a template for actions that change the structure of a design space represented as a DSN. Designs are selected at multiple levels: designers choose operators with which to create new designs, firms choose designs to realize as products, customers choose products to adopt, and investors choose firms to support with additional capital for growth. Although this stylized approach abstracts away from much of the richness of real-world innovation and product development processes, it can be used to build both simple and sophisticated models of architectural strategy in evolving complex systems.

Value plays a role in the theory of design evolution analogous to that of fitness in biological evolution: it is, by definition, the basis upon which selection occurs.<sup>15</sup> I follow the literature on network games, surveyed by Jackson (2004), in modeling the value realized by a designer in a given period as a function of the state of a network in that period—in this case, a design structure network. Value functions encode the modeler’s assumptions about how the observable features of a system design, including its inheritance and dependence relationships, affect the value of its constituent products to consumers, and how this value flows back to product owners in the form of profit. If there is only one designer, the problem of maximizing its next-period profit reduces to a standard dynamic programming problem. In the presence of multiple designers, the problem can be expressed as a noncooperative game (cf. Osborne and Rubinstein 1994, ch. 1).

---

<sup>15</sup>This formulation has been criticized as tautological, but it is not. Biological fitness may not be directly observable, but it refers to “the existence of certain survival-favoring attributes” that are conceptually distinct from the probability of survival in a given environment (Mayr 2001, p. 118). It is similarly difficult to observe economic value as perceived by consumers, but in the context of firms we can usefully equate it with profit or shareholder returns (Nelson and Winter 1982).



## 6 Beyond game theory: A case for computation

A model of strategic system design need not be fully realized in mathematical form, let alone rigorously validated, to be useful as an interpretive lens for real-world phenomena. Consider the Apple–Intel example again. Section 2 suggested moves that could be made by Apple, Microsoft, and Intel based on visual inspection of the DSNs in figures 1 and 2, and some elementary reasoning about economic forces in the PC industry.

Armed with the concepts of a system design game and the modular operators of Baldwin and Clark, we could go further in identifying general patterns of design evolution traced out by the observed design moves. For example, Apple’s switch from PowerPC to Intel processors was a straightforward act of porting.<sup>16</sup> The Boot Camp software, in contrast, involved at least three distinct architectural changes. Two of its components, the Windows device drivers and the BIOS emulation support, were created through porting, while the new boot loader was a result of substitution and inversion. The Boot Camp loader replaced the existing OS X boot loader (which is normally invisible to users) with one that provides a choice of operating systems, then invokes a normally hidden “hook” to continue the boot process.

To go still further requires a more formal approach that attempts to specify a particular game in quantitative or algebraic terms. There is nothing mysterious about this process—system design games are strictly a special case of the noncooperative games studied by game theorists and industrial economists. And indeed, simple SDGs can be studied using conventional analytic techniques. However, these techniques quickly reach their limits as one tries to extend the models to address more intricate questions of architectural strategy.

The limiting factor is neither the expressive power of the DSN formalism nor the ability to represent designers’ actions and incentives as modular operators and value functions, but rather the challenge of modeling the reasoning process of the designers in the absence of computable equilibria. Game-theoretic solution concepts like the Nash equilibrium help modelers by focusing attention on a narrow subset of decision rules that are consistent with the rational expectations of the players. Once we abandon the assumption of rational behavior, as we are forced to do by the complexity of real-world design problems, we are set adrift in a sea of possible decision rules that vary widely in their plausibility and appropriateness for a given model.

A sturdy life raft is provided by agent-based computational modeling techniques (Axtell 2000; Miller and Page 2007). Instantiating a system design game as an agent-based model transforms the modeler’s problem from a deductive one (i.e., what outcomes are *predicted* by a solution concept) to a constructive one (i.e., what outcomes are *observed* when computational agents are endowed with a decision rule). Computational experiments can efficiently explore a large space of decision rules and other model features, aided by modern distributed computing environments (e.g., clusters and grids) and active nonlinear exploration of the parameter space (Miller 1998).

---

<sup>16</sup>In fact, the core of OS X, called Darwin, was heavily based on the open-source FreeBSD operating system, which was originally designed for the Intel architecture and ported to the PowerPC by Apple. To avoid complicating the story, I have suppressed the fact that OS X continues to support the PowerPC architecture through its Universal binaries and Rosetta translation engine.

## 7 Conclusion

Traditional engineering representations such as software architecture diagrams and design structure matrices capture important features of complex system designs, particularly the structure of interdependencies among their components. Architectural strategy, however, requires thinking about the range of designs that are possible, not merely those that exist. A rigorous approach to architectural strategy therefore requires a richer formalism than ordinary networks of nodes and links. In this paper I proposed a new representation called a design structure network that graphically summarizes a system's design space.

DSNs express interdependencies among families of components that are related to each other by a type hierarchy. The primitive elements of a DSN include design spaces, components, types, and relationships of dependence and inheritance. I showed that DSNs capture familiar properties of systems in a natural way. For example, the information defining a type can be interpreted as an interface, yielding a dual relationship between types (sets of designs) and interfaces (visible information exposed by the members of such sets). This relationship, in turn, enables us to formalize a certain kind of compatibility between components: two components are *congruent* with respect to a third if they are interchangeable from its perspective, i.e., if the third is oblivious to the visible differences between the designs of the other two.

The DSN formalism, while parsimonious, is highly expressive. It includes DSMs as a special case, as well as the design hierarchies defined informally by Clark (1985) and Baldwin and Clark (2000). It even allows overlapping types (what software engineers would call "multiple inheritance"), enabling it to represent systems with the semilattice structure described by Christopher Alexander in his classic essay, "A City Is Not a Tree" (1965). The only practical requirement is for the design of interest to have some kind of modular structure, or else its DSN representation would be a single monolithic component.

This work contributes to the science of design by incorporating economic concepts like complementarity and substitutability into the representation of system architectures, thereby enabling architectural representations to be used directly in economic models of strategic behavior by value-seeking designers. Such models may be employed by system architects to anticipate the effects of their design moves on the competitive landscape in their industries, helping them "use architecture to win technology wars," to paraphrase Morris and Ferguson (1993). Models of strategic system design can also contribute to our knowledge of technological innovation, industrial organization, and industry evolution. This kind of research can thus serve as a bridge between the design-science and behavioral-science paradigms identified by Hevner et al. (2004).

## References

- Alexander, C. 1965. A city is not a tree. *Architectural Forum* 122:58–62.
- Amram, M., and N. Kulatilaka. 1999. *Real Options: Managing Strategic Investment in an Uncertain World*. Harvard Business School Press.
- Arthur, W. B. 1989. Competing technologies, increasing returns, and lock-in by historical events. *Economic Journal* 99(394):116–131.
- Axtell, R. 2000. Why agents? On the varied motivations for agent computing in the social sciences. CSED Working Paper 17, Brookings Institution.
- Baldwin, C. Y., and K. B. Clark. 2000. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press.
- Bass, L., P. Clements, and R. Kazman. 1998. *Software Architecture in Practice*. Addison-Wesley.
- 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.
- Cai, Y., and K. J. Sullivan. 2005a. Modeling and analysis of design space structures. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press.
- . 2005b. A value-oriented theory of modularity in design. In *EDSER '05: Proceedings of the 7th International Workshop on Economics-Driven Software Engineering Research*. ACM Press.
- Cardelli, L. 1997. Type systems. In *The Computer Science and Engineering Handbook*, ed. A. B. Tucker, 2208–2236. CRC Press.
- Clark, K. B. 1985. The interaction of design hierarchies and market concepts in technological evolution. *Research Policy* 14(5):235–251.
- Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. 2002. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley.
- Eppinger, S. D., D. E. Whitney, R. Smith, and D. Gebala. 1994. A model-based method for organizing tasks in product development. *Research in Engineering Design* 6(1):1–13.
- Farrell, J., and G. Saloner. 1992. Converters, compatibility, and the control of interfaces. *Journal of Industrial Economics* 40(1):9–35.
- Fowler, M. 2003. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd ed. Addison-Wesley.
- Garlan, D., and M. Shaw. 1996. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

- Grove, A. S. 1996. *Only the Paranoid Survive: How to Exploit the Crisis Points that Challenge Every Company*. Currency.
- Hevner, A. R., S. T. March, J. Park, and S. Ram. 2004. Design science in information systems research. *MIS Quarterly* 28(1):75–105.
- Holland, J. H. 1992. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. 2nd ed. MIT Press.
- Jackson, D. 2002. Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology* 11(2):256–290.
- . 2003. Module dependences in software design. In *Post-workshop Proceedings of the 2002 Monterey Workshop: Radical Innovations of Software and Systems Engineering in the Future*. Springer Verlag.
- Jackson, M. O. 2004. A survey of models of network formation: Stability and efficiency. In *Group Formation in Economics: Networks, Clubs, and Coalitions*, ed. G. Demange and M. Wooders, 11–57. Cambridge University Press.
- Katz, M. L., and C. Shapiro. 1994. Systems competition and network effects. *Journal of Economic Perspectives* 8(2):93–115.
- Krutchen, P. B. 1995. The 4+1 view model of architecture. *IEEE Software* 12(6):42–50.
- Langlois, R. N., and P. L. Robertson. 1992. Networks and innovation in a modular system: Lessons from the microcomputer and stereo component industries. *Research Policy* 21(4): 297–313.
- MacCormack, A., J. Rusnak, and C. Baldwin. 2006. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science* 52(7):1015–1030.
- Mayr, E. 2001. *What Evolution Is*. Basic Books.
- Merton, R. C. 1998. Applications of option-pricing theory: Twenty-five years later. *American Economic Review* 88(3):323–349.
- Miller, J. H. 1998. Active nonlinear tests (ANTs) of complex simulation models. *Management Science* 44(6):820–830.
- Miller, J. H., and S. E. Page. 2007. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton University Press.
- Morris, C. R., and C. H. Ferguson. 1993. How architecture wins technology wars. *Harvard Business Review* 71(3):86–95.
- Nelson, R. R., and S. G. Winter. 1982. *An Evolutionary Theory of Economic Change*. Harvard University Press.

- Northrop, L., P. Feiler, R. P. Gabriel, J. Goodenough, R. Linger, T. Longstaff, R. Kazman, M. Klein, D. Schmidt, K. Sullivan, and K. Wallnau. 2006. Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University.
- Osborne, M. J., and A. Rubinstein. 1994. *A Course in Game Theory*. MIT Press.
- Parnas, D. L. 1972. On the criteria to be used for decomposing systems into modules. *Communications of the ACM* 15(12):330–336.
- Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.
- Pimmler, T. U., and S. D. Eppinger. 1994. Integration analysis of product decomposition. In *DTM '94: Proceedings of the ASME 6th International Conference on Design Theory and Methodology*, 343–351.
- Pugh, E. W., L. R. Johnson, and J. H. Palmer. 1991. *IBM's 360 and Early 370 Systems*. MIT Press.
- Sangal, N., E. Jordan, V. Sinha, and D. Jackson. 2005. Using dependency models to manage complex software architecture. In *OOPSLA '05: Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, 167–176. ACM Press.
- Shannon, C. E. 1948. A mathematical theory of communication. *Bell System Technical Journal* 27:379–423, 623–656.
- Shapiro, C., and H. R. Varian. 1999. *Information Rules: A Strategic Guide to the New Economy*. Harvard Business School Press.
- Sharman, D. M., and A. A. Yassine. 2004. Characterizing complex product architectures. *Systems Engineering* 7(1):35–60.
- Shaw, M., and P. Clements. 2006. The golden age of software architecture. *IEEE Software* 23(2):31–39.
- Simon, H. A. 1996. *The Sciences of the Artificial*. 3rd ed. MIT Press.
- Steward, D. V. 1981. The design structure system: A method for managing the design of complex systems. *IEEE Transactions on Engineering Management* 28(3):71–74.
- Woodard, C. J. 2006. Architectural strategy and design evolution in complex engineered systems. Ph.D. thesis, Harvard University.