An Interdisciplinary Perspective on Interdependencies

Cleidson de Souza
University of California, Irvine
cdesouza@ics.uci.edu

David Redmiles
University of California, Irvine
redmiles@ics.uci.edu

May 2005

ISR Technical Report # UCI-ISR-05-7

The Interdisciplinary Study of Interdependencies

Cleidson de Souza[1,2]
[1]Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
cdesouza@ics.uci.edu

David Redmiles[1]
[2]Departamento de Informática
Universidade Federal do Pará
Belém, PA, Brazil – 66075
redmiles@ics.uci.edu

ISR Technical Report # UCI-ISR-05-7

May 2005

Abstract:

Modularity is a very general principle for managing complex systems. It suggests the division of a system into smaller parts, called modules. This principle has been applied to complex systems in many domains, including product engineering, organizations and development processes. In any domain, modules must interact in a coordinated fashion for an effective system. Interactions imply interdependencies. Therefore, interdependencies between the modules need to be supported—analyzed, engineered, documented, developed, and managed—efficiently and effectively if the complex system is to be successful. Indeed, different disciplines have created approaches for supporting interdependencies, each with its particular perspective. This paper surveys approaches from several different disciplines, including software engineering, organization science, management, computer-supported cooperative work, and human-computer interaction. It uses a theoretical framework that supports the comparison of approaches across disciplines. This paper also creates a vocabulary for discussing interdependencies. Commonalities among the approaches are identified and suggestions for promising research areas in the study of interdependencies are described.

The Interdisciplinary Study of Interdependencies

Cleidson de Souza[1,2]
[1]Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3425
cdesouza@ics.uci.edu

David Redmiles[1]
[2]Departamento de Informática Universidade Federal do Pará
Belém, PA, Brazil – 66075
redmiles@ics.uci.edu

ISR Technical Report # UCI-ISR-05-7

May 2005

Abstract:

Modularity is a very general principle for managing complex systems. It suggests the division of a system into smaller parts, called modules. This principle has been applied to complex systems in many domains, including product engineering, organizations and development processes. In any domain, modules must interact in a coordinated fashion for an effective system. Interactions imply interdependencies. Therefore, interdependencies between the modules need to be supported—analyzed, engineered, documented, developed, and managed—efficiently and effectively if the complex system is to be successful. Indeed, different disciplines have created approaches for supporting interdependencies, each with its particular perspective. This paper surveys approaches from several different disciplines, including software engineering, organization science, management, computer-supported cooperative work, and human-computer interaction. It uses a theoretical framework that supports the comparison of approaches across disciplines. This paper also creates a vocabulary for discussing interdependencies. Commonalities among the approaches are identified and suggestions for promising research areas in the study of interdependencies are described.

List of Figures

1 Introduction

Everyday it is possible to witness technological advances that benefit people. However, such advances should not be taken for granted. The effort to build products and enable services is tremendous and requires the coordinated efforts from hundreds or thousands of people. Different aspects (constraints, goals, requirements, etc) need to be handled, making this design process extremely complex.

One way to manage complexity is to decompose systems into subsystems (Simon 1996). This has been known as modularity. Basically, modularity means that a complex system can be decomposed into smaller pieces called *modules*. Modularity reduces complexity because it allows (i) one to deal with the details of each part of the system in isolation (by ignoring the details of other modules) and (ii) one to deal with the overall characteristics of all modules and their relationships in order to integrate them into a coherent system (Ghezzi, Jazayeri et al. 2003). The principle of modularity has been applied to many aspects of human endeavor. In software production, by creating decomposable systems, in organizations by creating group units and teams, in product development by creating subassemblies, and in *processes* of product development by creating tasks (Mintzberg 1979; Langlois 1999; Scott 2003). In any case, modules are parts of a large system and need to interact in some coordinated way for an effective system. Interactions between modules are necessary for the exchange of information, energy, and/or data. These interactions imply that these modules are interdependent[1]. In order words, modules rely or depend upon other modules to create the larger system. These interdependencies make it easier or more difficult to understand, extend, modify, and reuse these different modules. Therefore, their study is of fundamental importance for product, organization, and process design. Indeed, interdependencies – and dependencies - have been long studied in organization science (Thompson 1967), software engineering (Stevens, Myers et al. 1974), and computer-supported cooperative work (Malone and Crowston 1994). Each one of these disciplines, however, brings its own methods, perspectives, and assumptions to the study of interdependencies. By doing that, it is possible to advance each one of those fields, but it makes very difficult to "reuse" or adapt results across fields. A unifying framework able to summarize this diverse body of knowledge is missing. This framework would allow the identification of common themes among these diverse disciplines, but, more importantly, it allows one to identify research topics potentially useful in different disciplines.

The fundamental goal of this paper is exactly to propose this framework. This framework synthesizes arguments from a sample of approaches surveyed[2] from several different disciplines, including software engineering, organization and management sciences, computer-supported cooperative work, and human-computer interaction. It is intended to create a discipline-independent vocabulary for discussing dependencies. Paraphrasing Malone and Crowston (1994) when defined coordination theory: "by summarizing this diverse body of knowledge in a way that emphasizes its common themes, we hope to help to define a community of interest and to suggest useful directions for future progress". Similarly, I expect to help researchers and practitioners to identify open research questions and motivate them to answer them. To the best of my knowledge, no similar work can be found in the literature.

The proposed framework is based on three models describing the possible entities that are part of a dependence[3] relationship: products, tasks, and organizations. More importantly, the

[1] Interdependencies are bi-directional relations: A depends on B *and* B depends on A. Dependencies, on the other hand, are one-way relations: A depends on B.
[2] Because the survey of all existing approaches is impossible, a representative set was chosen based on their importance and accessibility.
[3] Note that dependency and dependence mean the same thing. In this paper, I will use the second term.

framework allows relationships between these models, (e.g. how the product and the task models are associated?). The three models are described in this paper, but only the product and the task models are discussed in details. Consequently, only relationships between these two models are described in this paper. Indeed, approaches describing these relationships are particularly important because they suggest that these models are interdependent. Changes in one model cause changes in the other and vice-versa. In addition, I also describe "common themes" found across models that grew out of the analysis of the approaches surveyed. They include the degree, value, and representation of dependencies. Finally, suggestions for future research are indicated.

A note about the strategy used in this survey is necessary at this point. Initially, I surveyed the existing literature and the web for theoretical studies of dependencies (e.g., Thompson 1967) as well as empirical studies of dependencies, particularly in software development settings (e.g., Grinter 2003). Particularly in organizational science, dependencies are usually linked to the coordination mechanisms needed to manage them, thus I also briefly discuss such mechanisms. After surveying the approaches existing in different disciplines, I created the framework proposed in this paper as an explanatory mechanism to locate and contrast them. As consequence, in the next step I identified "common themes" studied in these different approaches. Finally, I identified topics that I believe are interesting for future research.

The rest of this paper is organized as follows. The next section describes the theoretical framework used in this paper. Then, sections 3 and 4 describe the product and task models, respectively. After that, section 5 describes approaches that describe both product and tasks dependencies. Approaches integrating these models are presented in section 6. This discussion is followed by the common themes identified in different models. After that, section 8 suggests directions for future research on dependencies. Finally, conclusions and ideas for future work are presented.

2 Background and Theoretical Framework

2.1 What is dependence?

I surveyed the literature to understand how different disciplines study dependence relationships. Disciplines studied include organization and management sciences, software engineering, computer supported cooperative work, and human computer interaction. Based on this survey, it is easy to notice that different disciplines adopt distinctive definitions of dependencies. For example, in management and organization sciences, Crowston (2003) discusses how other authors see dependencies as relationships between organizations: "Litwak and Hylton (1962), for example, define interdependency as when two or more organizations must take each other into account if they are to accomplish their goals." In these same fields, other researchers define interdependencies as relationships between tasks: "[the] degree to which two or more activities interact to determine an outcome jointly" (Sorenson 2003). Finally, some authors define dependence as a relationship between two actors: define interdependence as "the extent to which unit personnel are dependent upon one another to perform their individual jobs" (Van de Ven, Delbecq et al. 1976).

Furthermore, even in the same discipline different definitions of dependence can be identified. For instance, in software engineering, Grinter (2003) broadly defines dependencies as "technical relationships in the code", while Wilde (1990) is more specific defining a dependency as "a relationship between two components so that changes to one may have an impact that will require changes to the other". Nevertheless, most authors define dependencies in specific contexts: *program* dependencies are "syntactic relationships between the statements of a program that represent aspects of the program's control and data flow" (Ferrante, Ottenstein et al. 1987; Podgurski and Clarke 1989); *component* dependencies reflect the potential for one component to affect or be affected by the elements (e.g., other components) that compose the system (Vieira 2003), and finally, *architecture* dependencies focus on "both the structural and behavioral relationships among components expressed in (…) formal architecture description languages" (Stafford, Wolf et al. 1998).

Based on these examples, it is not difficult to notice that (inter)dependencies have been looked at from different perspectives in the literature, including dependencies between social actors, products, organizations, and tasks. The underlying concept among these approaches is the same, though: *a dependence is a relationship between two entities that exists because one must interact with the other to accomplish something "larger" than the entities themselves*. Interdependencies between modules are a consequence of the need for combining these modules. For example, the different parts of a climate control system need to be integrated, but not randomly: the engine fan needs to provide airflow across the radiator, as a result the radiator is dependent on the engine fan (Pimmler and Eppinger 1994). Similarly, two classes in an object-oriented application are interdependent because they need to interact (through method calls) to produce the application behavior. Two tasks are interdependent if the actors performing them need to interact to determine an outcome jointly[4] (e.g., fixing bugs in the code and regression testing the new code both contribute to the larger outcome of developing high-quality software) (Sorenson 2003). Departments of an organization are also interdependent because both are parts of the same organization, therefore concerted action might be required to achieve organizational goals. Finally, two organizations are interdependent if they need to interact to exchange resources (e.g., resource-dependence theory). For the purposes of this paper, this is the definition of interdependence that I will adopt.

Note that a dependence is not a property of the entities themselves, but rather a relationship between them. Note also that a *consequence* of this relationship is that one entity might need to be changed, in case the other is changed. This distinction between the dependence and its consequence is particularly important since some authors use this consequence to define a dependency.

After surveying the approaches existing in different disciplines according to the definition presented, I created a theoretical framework as an explanatory mechanism to locate and contrast them. This framework allowed me to identify "common themes" in these approaches. The next section presents such framework.

2.2 Theoretical Framework

As mentioned before, several definitions of dependency have been proposed. Some describe dependencies between products, other as relations between organizations, and finally between tasks. To account for these different approaches, the framework proposed defines three models, namely products, organizations and tasks. Each model describes the set respective set of approaches. For instance, the product[5] model describes the approaches to the study of dependencies within a product or between products. Similarly, the organization (task) model describes approaches studying dependencies between organizations (tasks). In other words, the models of the framework are used to group similar approaches. The existence of these models also allowed me to study multi-model approaches, such as workflow management systems that can be used to describe dependencies between tasks, products and even organizations

[4] Note that social actors are interdependent to the extent that the *tasks* they are performing are interdependent. That means I am purposefully excluding more general sociological theories, such as Emerson's power-dependence relations (Emerson, 1962).

[5] The name product model is chosen to emphasize the broader scope of this work, including not only software engineering research but also hardware product research. Therefore, a product means a software and/or hardware artifact to be built by the organization.

(Shrivastava and Wheater 1999). The last advantage of these models is to facilitate the understanding of approaches that *link* them. For instance, Grinter's studies of recomposition suggest that product dependencies create task dependencies (2003). Each one of the models will be discussed below.

The *product model* describes dependencies between products or artifacts being developed by an organization. More specifically, dependence relationships either between parts of the same product or between different products. This model focuses on the structure of the artifacts and how this structure can be changed in order to manage – minimize or localize – the dependencies in the product. In addition, it emphasizes the impact that changes in one product will cause in other products. Examples of approaches located on the product model include program (Ferrante, Ottenstein et al. 1987), component (Vieira 2003) and architecture (Zhao 1997) dependencies, and traceability approaches that relate the different artifacts created during the software development process (Spanoudakis and Zisman 2004).

The second model, the *organizational*, focuses on how entire organizations interact in order to properly accomplish their goals. To quote Crowston (2003): "Litwak and Hylton (1962), for example, define interdependency as when two or more organizations must take each other into account if they are to accomplish their goals." The overall focus is on dependencies between entire organizations or between organizations and their environment, in accordance with the open system's view of organizations that recognizes the importance of organization-environmental connections, in contrast to the closed system view, which assumes that all elements and processes of interest in an organization are internal to this organization (Scott 2003). For instance, the resource-dependence approach (Scott 2003) is based on the view that the need for acquiring resources creates interdependencies between organizations. Furthermore, the importance and scarcity of these resources establishes the degree of organizational dependence. According to this approach, organizational members, naturally aware of these dependencies, will carefully choose suppliers and negotiate agreements with them in order to avoid "crippling dependencies" (Scott 2003, pg. 199). The degree of dependence between two organizations is a function of the importance of a resource given by the dependeer organization and the extent to which it is controlled by the dependee. The work surveyed in organizational dependencies can be translated into dependencies between elements of the other models, because organizations, for practical purposes, are composed of their personnel, tasks, products among other things (Scott 2003, pg. 18-24). In this case, the above mentioned resource-dependence approach can then be "translated" into product dependencies: one product (or task) in one organization requires another product from another organization. The other reason that led me to not focus on the organizational model is my research interest in the practices, the ways by which social actors manage their interdependencies. Having said that, I will not discuss the organizational model anymore in this work.

Finally, the *task model* is the next model of the framework. It describes how tasks or activities interact to determine an outcome jointly (Sorenson 2003). This model has also been largely studied in organization science. The focus is not only on the study the tasks performed by particular organizations, but also how the degree of interdependencies between tasks influences the choice of the associated coordination mechanisms to manage these interdependencies. As Thompson (Thompson 1967, pg. 55-56) simply puts:

> "In a situation of interdependence, concerted actions come about through coordination, and if there are different types if interdependencies, we would expect to find different services for achieving coordination."

Approaches in this model have been largely influenced by Thompson's work. Indeed, a 1995 survey of interdependencies - in the context of managerial and organizational sciences - identified the fundamental impact of Thompson's work in the literature of interdependencies (Staudenmayer 1997). Additional approaches include activity-based design structure matrices

(Eppinger 2001) and the coordination theory proposed by Malone and Crowston (1994). This model also includes work interdependencies between departments created by project tasks and the coordination mechanisms necessary to deal with them (Adler 1995).

Figure 1 below presents the framework and the associated models. Each department in the organization contains its own set of tasks, since these departments usually have different goals. Every model in the framework contains its own types of elements: the product model contains products (squares on the figure) and their dependencies, the task model contains tasks (circles) and dependencies between them, and so does the organizational model. Note that some relationships between the models are also indicated. For instance, products are developed by tasks.

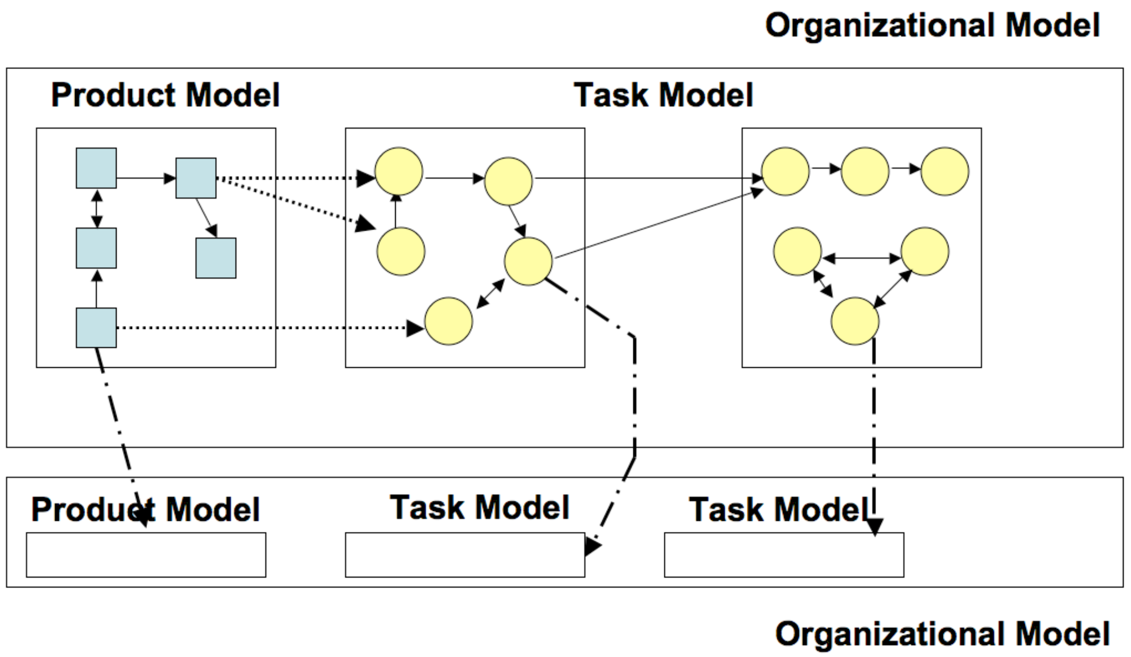


Figure 1 – Theoretical Framework and Models

Furthermore, these models influence each other. For instance, different studies have shown that dependencies in the product model create dependencies among actors who, therefore, need to communicate and coordinate to perform their tasks (Grinter 2003; de Souza, Redmiles et al. 2004). Similarly, interdependencies in the task model influence how organizations structure themselves (Mintzberg 1979).

The rest of this document focuses on the product and task models, because my overall research goal is to understand how software development dependencies influence –or are influenced by- the tasks that need to be performed in an organization. That is, how the product and the task models interact. As a result, relations of dependencies between two organizations are out of scope of this work.

3 The Product Model

One of the richest domains for studying dependencies in the product model is the domain of software development. The main reason being the fact that software is flexible enough to be easily changed when compared with hardware products. Thus, in software development new dependencies might be created (or removed) with minimum effort. Dependencies in this field are relationships between software development artifacts (e.g., requirements specifications, design models, and, more frequently, source code). Examples include program dependencies as syntactic relationships between the statements of an unique program (Ferrante, Ottenstein et al. 1987; Podgurski and Clarke 1989), and dependence traceability relationships, that establish trace relations between different artifacts (Spanoudakis and Zisman 2004). Research on this area has developed algorithms and approaches for dealing with these dependencies, even quantifying them through metrics (cohesion and coupling) and elaborated representations based on graphs. All these aspects are discussed in this section.

Despite the main focus on dependencies in software engineering, I will also present a technique that can be used to describe dependencies in any product development including development of automobiles, airplanes engines, and so on.

3.1 Program Dependencies

The first place to look at dependencies in software engineering is in the source code of a software application (or in programs, as some authors refer). Ferrante and colleagues (1987) describe two very simple examples that clearly illustrate the types of dependencies that occur in a program:

> "In the first case, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. For example, given the two statements S1 and S2 below:
>
> ```
> A:=B*C      (Sl)
> D:=A*E+l    (S2)
> ```
>
> S2 depends on Sl, since executing S2 before Sl would result in S2 using an incorrect value for A. Dependencies of this type are *data dependencies*. Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement. In the sequence below with statements S3 and S4
>
> ```
> if  (A)  then    (S3)
>       B:= C*D;   (S4)
> endif
> ```
>
> S4 depends on predicate (A) since the value of (A) determines whether S4 is executed. Dependencies of this type are called *control dependencies*."

Interest in control and data dependencies is specially important in compiler research because in order to perform program optimizations, it is necessary to be aware of these dependencies avoiding semantic changes to the program (Aho, Sethi et al. 1986). Program dependencies, in particular data flow dependencies, have been used by software engineers to improve software testing by defining code coverage criteria, that is, rules used for selecting data intended to ensure that certain portions of a program's code are "covered" or "exercised" (Podgurski and Clarke 1989). Other usages of program dependencies include fault location when debugging through program slicing (Weiser 1984), maintenance, parallelization, computer security, and code optimization (Podgurski and Clarke 1989).

In order to explicitly represent and manipulate both types of program dependencies, software engineering and programming language researchers adopt a common approach using program dependence graphs (or PDGs) (Ferrante, Ottenstein et al. 1987). According to Horwitz and Reps (1992), formally, a PDG for a program P is a directed graph whose vertices are connected by several kinds of edges. The vertices represent the statements of P, while the edges represent control and data dependencies, which are explained below:

> "The intuitive meaning of a control dependence edge from vertex v to vertex w is the following: if the program component represented by vertex v is evaluated during program execution and its value matches the label on the edge, then, assuming that the program terminates normally, the component represented by w will eventually execute; however, if the value does not match the label on the edge, then the component represented by w may never execute. (…) Data dependence edges include both flow dependence edges and

def-order dependence edges. Flow dependence edges represent possible flow of values, i.e., there is a flow dependence edge from vertex v to vertex w if vertex v represents a program component that assigns a value to some variable x, vertex w represents a component that uses the value of variable x, and there is an x-definition clear path from v to w in the program's control-flow graph."

For simplicity purposes, researchers initially explored the construction of PDGs for simple programs, either isolated procedures or programs that contain a single procedure. Later, interprocedural approaches were explored considering several procedure calls and their parameters and return types (Aho, Sethi et al. 1986). In this case, some authors adopt the term system dependence graph (SDG) instead of PDG. A SDG is made up of a collection of procedure dependence graphs, which are essentially the same as the program dependence graphs defined above, except that they may include additional and interprocedural control and flow dependence edges to represent procedure calls (Horwitz and Reps 1992). System dependence graphs can be used to construct call graphs (Lakhotia 1993) that are used for interprocedural program optimization and program understanding (Murphy, Notkin et al. 1995). According to Callahan and colleagues, a call graph " (…) summarizes the dynamic invocation relationships between procedures. The nodes of the call graph are the procedures in the program. An edge (pl, p2) exists if procedure pl can call procedure p2 from some *call site* within pl. Hence, each edge may be thought of as representing some call site in the program"(1990).

Despite their different applications in software engineering, Murphy and colleagues found out that call-graph extractors used in software development tools might provide different results (Murphy, Notkin et al. 1998). The reason why call-graph extractors provide so different results compared to those used in compilers is that "software engineering tools place a different-and in some ways more relaxed-set of requirements on call graphs, since the call graphs are most often consumed by humans for the purpose of understanding." (*ibid.*)

3.2 Component-based Dependencies

Since MacIlroy (1968) initially proposed the idea of software components in 1968, several definitions of software components can be found, some of them even contradictory[6]. For the purposes of this work, it is sufficient to describe a software component as physical packing of executable software with a well-defined and published interface. The notion of software component also implies higher-level abstractions than the code that it encloses, which means that in addition to the typical dependencies existing in any program (see previous section), components have additional types of dependencies. For instance, *context dependence* is the term used to describe the component's need for services from the deployment environment (Szyperski 1998). This dependence can not be avoided because in order to be "pluggable", components need necessarily to be linked to the deployment environment (e.g., Sun's Enterprise JavaBeans and Microsoft's .NET) where they were created (Brown and Wallnau 1998). To summarize, "components comprise a collection of aspects that can reveal different forms of dependencies in distinct levels of abstraction, making more complex the precise identification of dependencies" (Vieira and Richardson 2002). Therefore, component dependence analysis is crucial to effectively perform maintenance, evolution, testing, debugging, and management of component-based systems (Vieira 2003).

Vieira's work (Vieira 2003) summarizes important elements of component dependence analysis. For instance, he defines a conceptual framework for component dependencies in which he describes the several types of dependencies that a component might have, including internal (e.g., data and control dependencies) and external (e.g., and inter-component dependencies and

[6] The interested reader might refer to Vieira (Vieira 2003) for an overview of these definitions.

resource dependencies like context dependencies (Szyperski 1998)). In order to properly represent all the types of dependencies surveyed, he defines a representation called deployable dependence descriptions (DDDs) that describes a component's dependencies. DDDs can be automatically generated based on information available about the component or specified by the software engineer. When the DDDs of all components in a system are composed, a component-based dependencies model (CBDM for short) is created. Operations in this module are applied to assess overall system properties or individual components' properties. Vieira's approach focuses on inter-component analysis, when most approaches focus on intra-component analysis. Cadena (Hatcliff, Deng et al. 2003) is another tool that supports inter-component analysis.

3.3 Architecture-based Dependencies

Software architectures allow the definition of a high-level structure of a software system by describing a system's components and how those components are interconnected. They might also describe how those components interact by describing the behavior of each component when it interacts with the other components. The description of software architectures can be done informally using box and arrow diagrams, or using formal software architecture description languages (ADLs for short) such as ACME (Garlan, Monroe et al. 1995), C2 (Taylor, Medvidovic et al. 1996), Rapide (Luckham, Kenney et al. 1995), and Wright (Allen and Garlan 1994). The advantage of using formal ADLs is the possibility of reasoning about properties of the architecture, for instance verifying for deadlock, component mismatches, safety and so on. In this section I am concerned in particular with architectural dependence analysis, that is, the task of identifying and analyzing the dependencies of a particular software architecture. These techniques operate on the formal architecture description of a software system instead of the source code adopted by program dependencies. They can be used to support architectural reuse, change impact analysis, regression testing, and software understanding.

Architectural-level dependence analysis was first described by Zhao (1997). He uses ACME (Garlan, Monroe et al. 1995) as the underlying architectural description language and proposes three types of dependencies among the components of the architecture, namely, component-connector dependencies, connector-component dependencies, and additional dependencies that describe dependencies within a connector or component. Zhao describes a software architectural dependence graph (SADG), similar to program dependence graph (PDG), that represents the dependencies between the architectural elements. Slicing techniques are then applied on this graph to facilitate architectural understanding and reuse.

Stafford and colleagues (Stafford, Wolf et al. 1998) also applied concepts from program dependence analysis to software architectures. They use RAPIDE (Luckham, Kenney et al. 1995) as the underlying architectural description language. Instead of adopting some form of graph representation like Zhao (1997), they represent the dependencies in a dependence matrix. Dependence types are based on behavioral properties of the architecture (e.g., temporal – the behavior of one component precedes or follows the behavior of another component) or on structural relationships that link components of the architecture with modules implementing them (e.g., textual inclusion – the specification for a component may be created from numerous source modules that are textually combined). In more recent work, Stafford (2001) describes Alladin, an architecture-level dependence analysis tool.

Note that ADLs play a fundamental role in architectural dependence analysis. The dependencies that can be established among components of the architecture are heavily influenced by the primitive features of the ADL chosen (Stafford, Wolf et al. 1998). For instance, event-based ADLs allow the description of *temporal* constraints among components introducing additional complexity to architectural dependence analysis compared to traditional program dependence analysis (Stafford and Wolf 2001). Dependence analysis was later incorporated into other software architecture tools, such as Argus-I (Vieira, Dias et al. 2000) and Cadena (Hatcliff, Deng et al. 2003).

3.4 Cohesion and Coupling

Cohesion is a property of a software module, or program, that represents the functional relatedness of their internal elements (e.g., statements, procedures, and declarations) (Stevens, Myers et al. 1974). It is a measure of the relationships between its elements. Ideally, "elements of a module are grouped together in the same module for a logical reason, not just by change: they cooperate to achieve a common goal, which is the function of the module." (Ghezzi, Jazayeri et al. 2003). That is, a module has high cohesion if all of its elements are related strongly, and low cohesion otherwise.

The internal cohesion of a module is measured in terms of the strength of binding elements within the module. Cohesion occurs on the scale of weakest (least desirable) to strongest (most desirable) in the following order (Constantine and Yourdon 1979): coincidental, logical, temporal, communication, sequential, functional, and informational. *Coincidental cohesion* occurs when the elements within a module have no apparent relationship to one another. This results when a large, monolithic program is "modularized" by arbitrarily segmenting the program into several small modules, or when a module is created from a group of unrelated instructions that appear several times in other modules. *Logical cohesion* implies some relationship among the elements of the module; for example, in a module that performs all input and output operations, or all error handling operations. A logically bound module often combines several related functions in a complex and interrelated fashion. *Temporal cohesion* implies that all components, which are activated at a single time (e.g., start up or shut down), are brought together. Modules with temporal cohesion exhibit many of the same disadvantages as logically bound modules. The elements of a module possessing *communicational cohesion* refer to the same input data or produce the same output data. For example, "print and punch the output file" is communicationally bound. *Sequential cohesion* of elements occurs when the output of one element is the input for the next element. *Functional cohesion* is a strong, and hence desirable, type of binding of elements in a module because all elements are related to the performance of a single function. Finally, *informational cohesion* of elements in a module occurs when the module contains a complex data structure and several routines to manipulate the data structure. Each routine in the module exhibits functional binding. Informational binding is the concrete realization of data abstraction. The difference between informational cohesion and communicational cohesion is that communicational cohesion implies the all code in the module is executed on each invocation of the module while the other requires that only one functionally cohesive segment of the module be executed on each invocation (Sommerville 2000).

It is important to mention that these cohesion classes are not strictly defined, they might even overlap, which means that sometimes it is not easy to decide under which cohesion category a program unit should be classified. Furthermore, it is also possible and desirable to extend this classification to accommodate new programming paradigms. Sommerville (2000) added a new class of cohesion related to object-oriented approaches:

> "*Object cohesion* – each operation provides functionality which allows the attributes of the object to be modified, inspected or used as basis for service provision."

Therefore, a cohesive object is one that models a single entity (concept or abstraction) and all the operations required to manage or deal with that entity are part of the object. That means that subclasses that inherit attributes and methods from their parent classes cannot be seen as a single-unit entity, therefore they have reduced coupling compared to classes that do not inherit methods from others. Of course, inheritance might reduce coupling, but it supports software reuse, another desired goal in software development.

While cohesion measures the degree of dependencies that occur within a module, *coupling* measures the relationships between different modules. The term *coupling* is used as a measure of the interdependencies between two modules (Stevens, Myers et al. 1974; Ghezzi, Jazayeri et al. 2003). If two modules depend on each other heavily by having strong interconnections, they are said to have high coupling, otherwise, the modules are said to have low coupling and are almost independent of each other.

Coupling between modules occurs under different situations. For instance, given two modules x and y, if the x refers to the inside of y (it branches into, change data in or alters a statement in y), then these modules are said to have *content coupling* (Fenton and Pfleeger 1997). Another example of coupling happens when modules make use of shared variables (*common coupling*) or interchange control information (*control coupling*) (Constantine and Yourdon 1979). *Stamp coupling* happens when an entire data structure is passed as a parameter to a called module that only uses some of the fields from this data structure. In *data coupling*, two modules are said to be data coupled if all arguments are homogeneous data items. If a data structure is passed then the called module must use all the fields. Data coupling is a desirable goal. Two modules which are data coupled are easier to maintain since changes in one are less likely to impact the other. *Loose coupling* is achieved by ensuring that details of the data representation are held within a component. Its interface with other components should be through a parameter list. If shared information is necessary, the sharing should be limited to those components, which need access to the information. To summarize I will describe a scale from less desirable to most desirable coupling: content, common, control, stamp, and data.

Coupling often implies that if one wants to reuse a software module, he or she will also have to import all the other modules with which it is coupled. That is, high coupling reduces the reusability of a software module. According to Stevens *et al*. (1974):

> "The fewer and simpler the connections between modules, the easier it is to understand each module without reference to other modules. Minimizing connections between modules also minimizes the paths along which changes and errors can propagate into other parts of the system, thus eliminating disastrous "ripple" effects where changes in one part cause errors in another, necessitating additional changes elsewhere, giving rise to new errors, etc."

The concepts of coupling and cohesion are interrelated, while cohesions refer to the degree of interconnection (or relatedness) among the parts of a single module, cohesion refers to the strength of the interconnections between different modules. Both concepts help establishing the quality of a particular design (Sommerville 2000). In fact, by designing modules with high cohesion and low coupling, maintainability is achieved. If it becomes necessary to change a system, the part to be changed is easily identified because it can be found in a single place. It is not necessary to modify different parts of the program if a change has to be made. In other words, ideally, modules with high cohesion and low coupling are easier to analyze, understand, modify, test or reuse them separately (Ghezzi, Jazayeri et al. 2003).

3.5 Software Traceability

Dependence relationships between software development artifacts have also been studied under the name traceability. Software traceability is defined as "the ability to relate *artifacts* created during the development of a software system to describe the system from different perspectives and levels of abstraction with each other, the stakeholders that have contributed to the creation of the artifacts, and the rationale that explains the form of the artifacts" (Spanoudakis and Zisman

2004). In a survey of the area Spanoudakis and Zisman (2004) identified seven possible types of relationships between software artifacts. Dependence is one of them. According to these authors, in software traceability, not all authors use the term dependence to refer to this type of relationship; other names used include causal conformance, developmental relations, and correspondence to name a few.

Traceability was first conceived to describe and follow the life of a requirement. Originally, called requirements traceability, "with its emphasis on supporting the customer in ensuring that the requirement agreed upon are met" (Lindvall and Sandahl 1996). As part of this emphasis, researchers have studied interdependencies between requirements. For instance, Carlshamre and colleagues (2001) conducted a survey in five different companies and found out that "each of the cases [companies] had roughly 20% singular requirements" (requirements that do not depend on others) and that "by identifying the 20% 'most dependent' requirements, one can cover between 67% and 79% of all dependencies". Other empirical studies suggest that requirements indeed do affect each other (Dahlstedt and Persson 2003) and can be found in a diverse range of software development organizations (Ramesh and Jarke 2001). These empirical data highlight the importance of handling requirements dependencies during the specification of software development projects. The interested reader might refer to Dahlstedt's and Person's work (2003), which reviews requirement interdependencies and summarize part of the work and models developed in the area.

While requirements dependencies involve a relationship between two artifacts of the *same* type, traceability dependencies also involve artifacts of *different* types[7]. Furthermore, these artifacts could potentially be created at different moments in the software development process. For instance, a dependence might exist between a requirement and the design diagram that realizes this requirement. Lindvall and Sandahl (1996) classify these approaches as vertical and horizontal traceability. Vertical relationships exist between the same type of artifacts, while horizontal relationship exist between different types of artifacts. Figure 2 below exemplifies this classification.

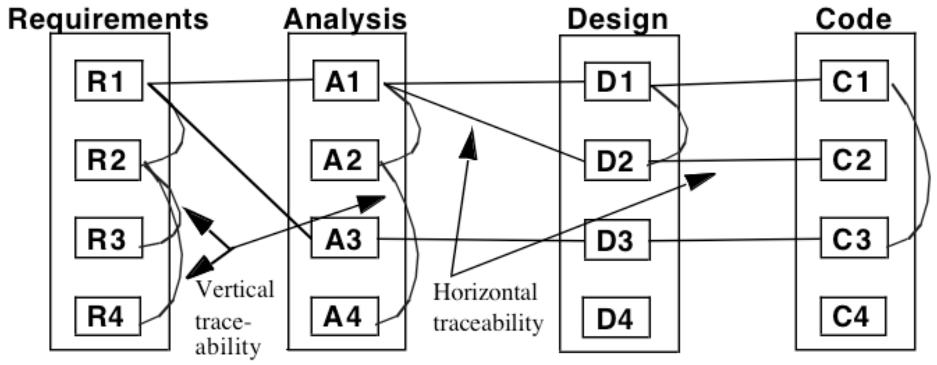


Figure 2 – Vertical and Horizontal Traceability (Lindvall and Sandahl 1996)

The existence of a dependence link between the requirements and the analysis and later to a design document that implements this requirement, is then, seen as something positive or beneficial, because it indicates that this particular requirement has been addressed by the software being implemented. Furthermore, some authors believe that requirements interdependencies can support software reuse: if similar requirements are identified when the stated requirements are compared with existing requirements, then this indicates a possible reusable component (Dahlstedt and Persson 2003).

[7] Program, component, and architecture dependencies also involve relationships between artifacts of the same type.

3.6 Component-based Design Structure Matrices

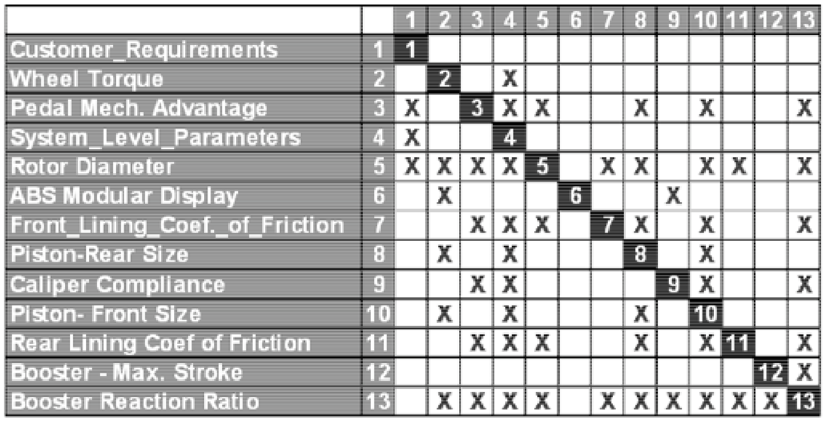
The design structure matrix (from now on called, DSM) is a tool that displays the relationships between elements of a system in a compact, visual, and analytically advantageous format (Browning 2001). Initially proposed by Steward (1981) to describe relationships in product development processes, it has been actually used to analyze complex like organizations, products and processes[8]. To model such diverse uses of DSM, Browning (2001) classified DSM in four main types:

- Component-based DSMs are used to model system architectures, or products based on components and/or their interdependencies;
- Team-based DSMs model organization structures according to people and/or groups;
- Activity-based DSMs model processes and networks of activities and the information flow among those activities; and
- Parameter-based DSMs model low-level relationships between design decisions and parameters, systems of equations, subroutine parameter exchanges, etc.

These four types of DSMs are aggregated into two main categories: static and time-based DSMs (Browning 2001). The distinction between them is based on the simultaneous existence of the elements. In static matrices, the elements of the DSM exist simultaneously. In time-based matrices the ordering of rows and columns indicate a flow through time so that some elements precede others: an element can only exist after the preceding element ceases to exist. Component and team-based DSMs are static and therefore analyzed with clustering algorithms. Activity and parameter-based DSMs are time-based and typically analyzed using sequencing algorithms. Despite these differences, concepts underlying the DSM approach are very similar and are described below.

Basically, a DSM is a square matrix where rows (and columns) describe the elements that compose a complex system. A mark X in row *i*, column *j* indicates that element *i* interacts with or is dependent on element *j*. The diagonal of the matrix is not used. Figure 3 presents a DSM describing decisions made for a brake system. It tells the reader that decisions about the "ABS Modular Display" (row 6) are affected by decisions made about the "Wheel Torque" (column 2). That is, inspecting a decision located in a particular row tells the reader which other decisions affect this one. Conversely, by reading down column 2 ("wheel torque") it is possible to find out it influences decisions about the "rotor diameter", "ABS modular display", "piston-rear size", "piston-front size", and "booster reaction ration" (rows 5, 6, 8, 10 and 13, respectively).

[8] In other words, DSM are used both on the product and task models.



		1	2	3	4	5	6	7	8	9	10	11	12	13
Customer_Requirements	1	1												
Wheel Torque	2		2		X									
Pedal Mech. Advantage	3	X		3	X	X			X		X			X
System_Level_Parameters	4	X			4									
Rotor Diameter	5	X	X	X	X	5		X	X		X	X		X
ABS Modular Display	6		X				6			X				
Front_Lining_Coef._of_Friction	7			X	X	X		7	X		X			X
Piston-Rear Size	8		X		X				8		X			
Caliper Compliance	9			X	X					9	X			X
Piston- Front Size	10		X		X				X		10			
Rear Lining Coef of Friction	11			X	X	X			X		X	11		X
Booster - Max. Stroke	12												12	X
Booster Reaction Ratio	13		X	X	X	X		X	X	X	X	X	X	13

Figure 3 - DSM describing decisions made for a brake system (Yassine 2004)

It is possible to replace the marks (X's) in the matrix by values quantifying the strength or degree of the dependence between two elements. For instance, Pimmler and Eppinger (1994) adopt a scale from -2 to +2 to indicate if a dependence is detrimental, neutral or required in a component-based DSM. Eppinger (2001) describes the frequency of team interactions in team-based DSMs ranging from daily to monthly interactions.

In both static and time-based matrices, the analysis is based on reordering the rows and columns of the matrix. Analysis of component-based DSMs are described below, while the analysis of activity-based DSMs are described in section 4.4.

In general, in component-based DSMs, the elements of the matrix that are used in the rows and columns are the modules that comprise the product being analyzed. For instance, the components of a car, computer, airplane engine, or software application. These DSMs are analyzed using clustering algorithms that aim to find "subsets of DSM elements that are mutually exclusive or minimally interacting" (Yassine 2004). The ultimate goal is to identify modules of a system that are as independent as possible.

Pimmler and Eppinger (1994) classify the dependencies between two components in four types:

- Spatial, the need for adjacency or orientation between two elements;
- Energy, a energy transfer between two elements;
- Information, the need for information or signal exchange between two elements; and
- Materials, the need for materials exchange between two elements.

Rather than having a single mark ("X") indicating the existence of the dependence between the elements, each intersection between a row and a column contains now another matrix of four elements describing the strength of the dependence according to these four types of dependence. Clustering can be applied according to one of these four dimensions and new insights might rise based of the four, possibly different, configurations of the product that the clustering algorithm will provide.

Baldwin and Clark (2000) adapted DSMs to make use of the information hiding principle (Parnas 1972). More specifically, a component of a DSM is separated into their interfaces and implementation parts. Interfaces are called design rules, while implementation parts are called hidden parameters. By separating the interface and implementation parts of a DSM component, they minimize the dependencies of the overall product improving its design. Interfaces are represented separately in these adapted DSMs to indicate their fundamental role. The resulting design structure matrix can then be clustered to generate modular designs. Figure 4 below presets a DSM without the usage of interfaces, while Figure 5 presents one that was changed trough the addition of interface *I* to minimize the dependence between components B and A. Both Figures are from (Sullivan, Griswold et al. 2001),

	A	B	C
A	.		
B	X	.	X
C		X	.

Figure 4 - A DSM without interfaces

	I	A	B	C
I	.			
A	X	.		
B	X		.	X
C			X	.

Figure 5 - A DSM with interfaces

Design-structure matrices have also been applied to software design. Sullivan et al. (2001) apply DSMs to software design and extend them to model environment variables in which the software is intended to be used. Their goal is to be able to account changes in the environment and their implications in the software design.

3.7 Discussion

Two main types of methods can be identified among the approaches dealing with dependencies in the product model: methods pursuing the analysis of product dependencies and methods pursuing the record of product dependencies. In the former case, I find program, component and architecture dependencies, as well as cohesion and coupling measures. The first three approaches share the usage of the program dependence graph or variations as representations for product dependencies. And, most importantly, they share the analytical goal of identifying and analyzing dependencies in the product to be able to answer questions about reusability, maintenance and understanding. In general, component and architecture dependencies are the evolution of program dependencies that arose of the need to deal with other abstractions (component and architectures) for building applications. Similarly, cohesion and coupling are measures of the degree of dependence within a module or between modules. That is, they require an analysis of the dependencies in one or more product to identify their intensity.

The second type of approaches is mostly associated with the record of dependencies so that they can be later used. Dependence relationships in software traceability are recorded to be later used to facilitate change impact analysis, maintenance and reuse approaches. The assumption underlying this approach is that recording dependencies is beneficial to the software development process, because this information aids the execution of some software development activities. In addition, in order to record dependencies it is necessary to adopt a representation for them (Spanoudakis and Zisman 2004).

The issue of identifying dependencies is more present on the first group of approaches where this task can be mostly automated since the structure of the software can be analyzed. One of the ultimate goals of traceability approaches is to be able to do this automatic identification (Spanoudakis and Zisman 2004).

4 The Task Model

In contrast to the software engineering literature, most work in organizational science has adopted the perspective of dependencies between tasks or activities. The main reason for that is the seminal work of Thompson (1967). Research on dependencies in this model is directly connected to work in *coordination mechanisms* for managing these dependencies. The concern with coordination practices occurs because tasks must, in some sense, be performed by actors, who will inevitably need to deal with these dependencies. Dependencies between tasks also influence an organization's structure, since, often, most organizations structure themselves to minimize the coordination required in their work (Mintzberg 1979).

4.1 Thompson's Work in Organizational Interdependencies

Thompson's (1967) work in interdependencies is an example of contingency models in organizational science. It is based on the open system's view that recognized the importance of organization-environmental connections, in contrast to the closed system view, which assumes that all elements and processes of interest in an organization are internal to it. Contingency theory:

> "(…) insists that there is no single best way in which to design the structure of an organization. Rather, what is the best or most appropriate structure depends -is contingent on- what type of work is being performed and on what environmental demands or conditions confront the organization." (Scott 2003)

In his work, Thompson divides an organization in three levels. The *technical* level is the one responsible for effectively carrying out the functions of the organization, transforming inputs into outputs. The *managerial* level is the part of the organization that designs and controls the technical level, by finding inputs, allocating personnel to the units, and so on. Finally, the *institutional* level is ensured with the relationships of the organization and the wider environment, establishing its domains, boundaries and legitimacy. Based on these organizational levels and on the insight of the open systems approach, Thompson argues that, to the extent possible, organizations try to seal off their core technology from environmental influences by adopting different *buffering* tactics. An example of such tactic is coding, which is the classification of inputs before inserting them into the technical core, such as quality control in assembly lines that inspect parts before they are placed in the production line and "triage" of patients according to the severity in emergency rooms (Scott 2003, pg. 199-200). Other tactics include stockpiling and leveling. Organizations need to seal off their technical core because the external environment might influence the organization. For instance, organizations depend on each other when they exchange resources (products, knowledge, information, personnel, and so on). In summary, "bridging tactics might be viewed as a response to –as well as a stimulus for – increasing organizational interdependence" (Scott 2003, pg. 203).

According to Thompson, three possible types of interdependencies can be identified among parts of an organization, or more specifically between tasks performed within an organization. *Pooled* interdependence links almost independent entities. Brokers, for example, provide this function. *Sequential* interdependence occurs in cases where serial dependence exists; assembly lines in a factory typically operate with this type of interdependence. Finally, *reciprocal* interdependence between two entities A and B covers cases where A affects B's output and B affects A's output. A reciprocal interdependency is like a baseball team, where each part focus on the whole for a while, then hands it off to another part, and gets back when this part's expertise is required (Mortensen and Hinds 2002). According to the naming scheme used in this paper, pooled tasks are independent, sequential tasks are dependent, and reciprocal tasks are interdependent.

Thompson theorized that a relationship exists between the type of interdependence and the coordination mechanisms[9] used. Pooled interdependencies are solved by using standardization of the work flow, sequential interdependencies are addressed using planning and scheduling, and finally, reciprocal forms of interdependencies require mutual adjustments. Moreover, he argues that organizations structure themselves in such a way to minimize the coordination costs required to perform. Since reciprocal interdependencies are the most costly, the lowest-level units will

[9] Coordination modes in Thompson's terms.

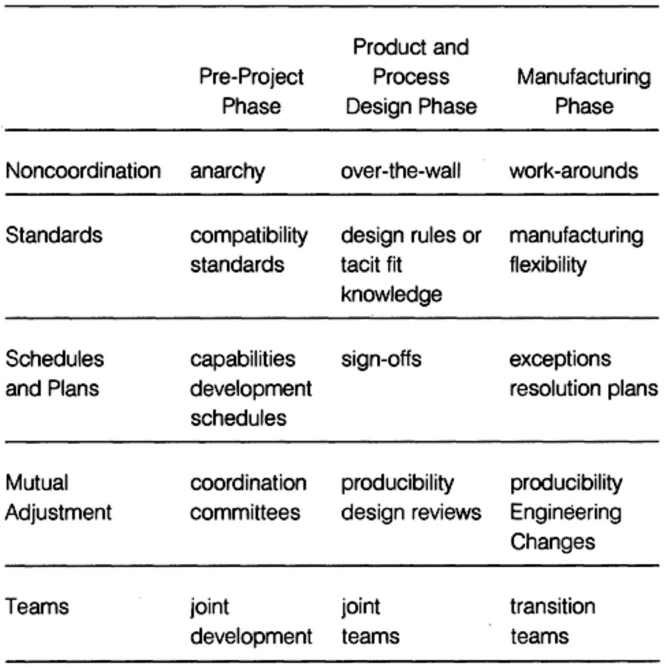
group to deal with them. Then, higher-groups are then formed to handle the remaining sequential interdependencies in the organization. Finally, groups, if necessary, are formed to handle any remaining pooled interdependencies. Note that, by doing that, organizations will create a hierarchy of units, similar to a process of successive clustering. Mintzberg (1979) calls this criteria for unit grouping in organizations *work-flow interdependencies*. This happens when groups of operating tasks are organized to reflect the sequence of tasks that need to be performed. The advantages of this approach are briefly described below:

> "members of a single unit have a sense of territorial integrity; they control a well-defined organizational process; most of the problems that arise in the course of their work can be solved simply, thought their mutual adjustment; and much of the rest, which must be referred up the hierarchy, can still be handled within the unity, by that single manager in charge of the work-flow."(Mintzberg 1979)

4.2 Interdepartamental Interdependencies

Paul Adler (1995) describes a field study where he focuses on the relationship between product design and manufacturing processes, i.e., how these two processes need to be coordinated to release new or improved products, avoiding designs that are not "manufacturable" or missing changes that would reduce the manufacturing cost. His work is based on field studies conducted in nine organizations that print circuit boards for electronic components and four organizations dealing with operations in hydraulic tubing for aircrafts. Adler's work focuses on interdepartmental interdependencies "created by project tasks thus, abstracting from interdependencies due to the broader organizational context in which the department operate." He adopts Thompson's ideas about coordination approaches to deal with interdependencies. In addition, he uses the *team* approach proposed by Van de Ven et. al. (1976)[10]. In addition, he also considers the progress of the project according to three phases: pre-project, product and process design phase, and manufacturing phase. However these phases are used for analytical purposes only, they are not as clearly defined in the organizations' daily work. As Adler points out, "the development of capabilities is an ongoing process, and both product and process design often evolve after the manufacturing release". Based on these four coordination mechanisms and project phases, he develops a taxonomy drawing on data from his field work. This taxonomy is described below:

[10] According to Van De Ven, the team approach "refers to situations where the work is undertaken jointly by unit personnel who diagnose, problem-sole and collaborate in order o complete the work". The team approach distinguishes from Thompson's mutual adjustment by the simultaneity of multilateral interactions and which typically requires physical proximity. There is no measurable time lapses in the work done by the team members.



	Pre-Project Phase	Product and Process Design Phase	Manufacturing Phase
Noncoordination	anarchy	over-the-wall	work-arounds
Standards	compatibility standards	design rules or tacit fit knowledge	manufacturing flexibility
Schedules and Plans	capabilities development schedules	sign-offs	exceptions resolution plans
Mutual Adjustment	coordination committees	producibility design reviews	producibility Engineering Changes
Teams	joint development	joint teams	transition teams

Figure 6 – A typology of design/ manufacturing coordination mechanisms (Adler, 1995)

This taxonomy describes which coordination mechanisms are used in each phase of the design/manufacturing process. It is later translated into a normative theory generated from his data. Accordingly, managers could use this normative theory to decide which coordination mechanisms to use in their project. This theory is summarized below (Figure 7) and basically argues that an increasing novelty of fit issues requires more "elaborated" coordination mechanisms, and, at the same time, a decreasing analyzability of fit issues requires more coordination effort in later phases of the project (Adler 1995, pg. 159). Finally, Adler discusses the cost – benefits tradeoffs that underlie his normative theory.

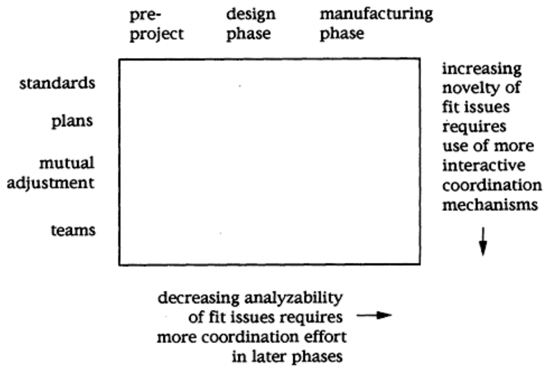

pre-project
design phase
manufacturing phase
standards
plans
mutual adjustment
teams
increasing novelty of fit issues requires use of more interactive coordination mechanisms
decreasing analyzability of fit issues requires more coordination effort in later phases

Figure 7 – Coordination Mechanisms vs. Phases of Process (Adler, 1995)

Adler's work is one of the few that takes into account temporal aspects. Note also that the temporal aspect of interdependencies arose from his data. As he describes:

> "(…) as the phases of work unfold within a time-bound project, departments typically experience different degrees and types of interdependencies, and they interact with varying intensities and via different coordination mechanisms. And as a result, in the course of a

product development project, neither interdepartmental interdependencies nor coordination mechanisms are constant over time."

However, Adler analyzes how *different* tasks require different coordination mechanisms over time, instead of how the *same* task might change its coupling with other tasks over time. I will discuss this in more details in section 8.

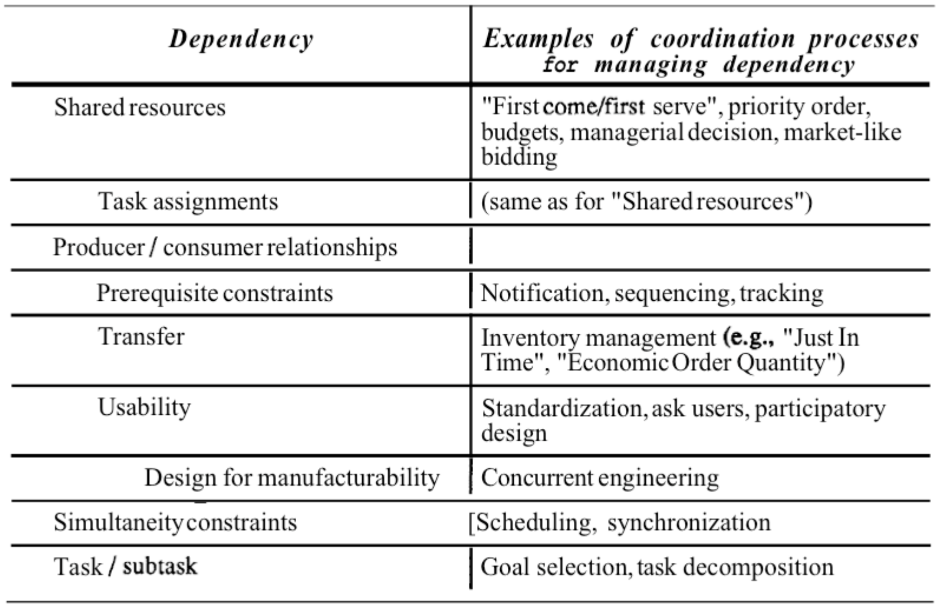
4.3 Coordination Theory

In an important paper in 1994 Malone and Crownston (1994) summarized the body of knowledge, at that time, about coordination under the name of *coordination theory*. Coordination theory draws results from different fields such as computer science, organization theory, economics, biology, sociology, social psychology, linguistics, law, and political science. In writing the paper, they had different goals, including to define a community of interest, to suggest useful directions for future research, and most importantly, to facilitate the "transport" of results back and forth between these different fields.

Coordination is defined as the management of interdependencies between activities, therefore acknowledging a long history of emphasizing dependencies in organization theory starting with Thompson (see section 4.1). Based on this definition, it is easy to note that interdependencies play a fundamental role in this theory. Indeed, this theory is based on study of the types of dependencies that might exist between two activities, and, consequently the types of coordination mechanisms used to manage them.

Interdependencies are seen as occurring between two activities or tasks. The authors argue that it is not necessary to study interdependencies between components or artifacts because they "explicitly or implicitly, affect the performance of some activities". Therefore, viewing interdependencies as solely occurring between activities simplifies the approach.

Interdependencies and associated coordination mechanisms are classified as: managing shared resources, managing producer/consumer relationships, managing simultaneity constraints, and managing task/subtask dependencies. In addition, two other activities are considered: group decision-making and communication. All these approaches are described in Figure 8.



Dependency	*Examples of coordination processes for managing dependency*
Shared resources	"First come/first serve", priority order, budgets, managerial decision, market-like bidding
Task assignments	(same as for "Shared resources")
Producer / consumer relationships	
Prerequisite constraints	Notification, sequencing, tracking
Transfer	Inventory management (e.g., "Just In Time", "Economic Order Quantity")
Usability	Standardization, ask users, participatory design
Design for manufacturability	Concurrent engineering
Simultaneity constraints	[Scheduling, synchronization
Task / subtask	Goal selection, task decomposition

Figure 8 – Dependency Types and associated Coordination Mechanisms

Based on this analysis of interdependencies, Malone and Crowston apply this coordination perspective in different situations, for instance in the design of cooperative support tools and to facilitate the understanding of the effects of information technology in organizations. Finally, they identify several aspects where coordination theory needs to be expanded, one of them being how to represent coordination processes. That is:

> "When should we use flowcharts, Petri nets, or state transitions diagrams? Are there other notations that are even more perspicuous for analyzing coordination? How can we classify different coordination processes? For instance, can we usefully regard some coordination processes as 'special cases' of others? How are different coordination **processes** combined when activities are actually performed?" [emphasis in the original]

Because in coordination theory, coordination mechanisms and interdependencies are so closely linked, it seems that the question about how to represent coordination mechanisms is closely related to the question of how to represent dependencies.

4.4 Activity-based Design Structure Matrices

Design structure matrices have been explained in more details previously in section 3.6. This section details activity-based DSMs.

An activity-based DSM is a matrix where rows (and columns) describe the tasks or activities that compose a particular development process of an organization[11]. For instance, Intel's semi-conductor development process described in (Eppinger 2001). Activity-based DSMs can be applied to:

> "(…) to organize the design of a system, to develop an effective engineering plan, to show where estimates are required, and to analyze the flow of information that occurs during the design work." (Steward 1981)

Activity-based DSMs are especially useful in concurrent engineering processes, where tasks are performed in *parallel*. They also allow one to identify *coupled* tasks (tasks that are interdependent), and *sequential* tasks (where one task requires input from the other task, but not vice-versa). Activity-based DSMs allow the *identification* of these "types" of tasks even if they have not yet been recognized in the organization. For example, tasks A and C in Figure 9 are sequential tasks. Tasks D and F, are said to be parallel if D's output is not used by F's input and vice-versa. Because they are not interdependent they can be performed concurrently. Finally, tasks E, G and I are coupled. This means that in order to perform task E, the outputs of tasks G and I are necessary. However, tasks G and I require E's output to be performed as well.

[11] Identifying the sequence of steps of a process is left out of the scope of the DSM. One might use interviews, surveys, and organizational documents to identify this process.

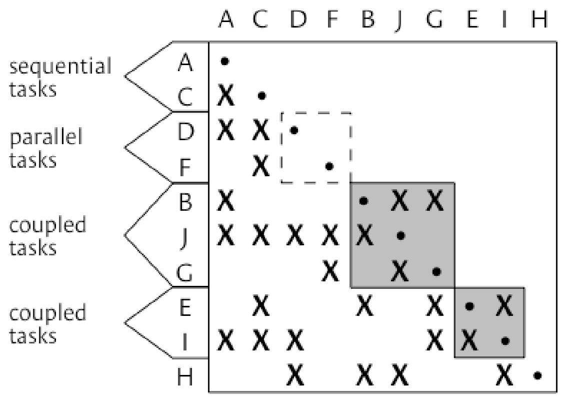


Figure 9 – Example of Design Structure Matrix after optimization

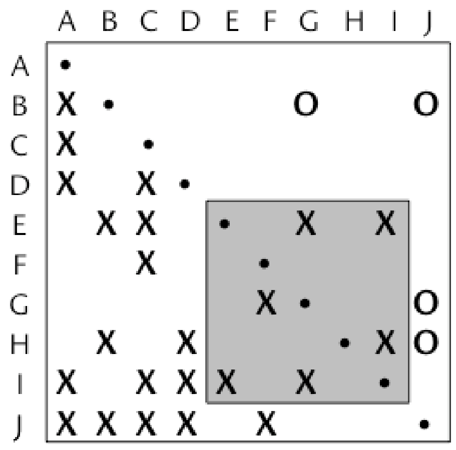


Figure 10 - Example of Design Structure Matrix before optimization

Interdependent tasks are easily identified in a activity-based DSM. Items below the main diagonal indicate tasks that require input from tasks already performed. On the other hand, tasks above the diagonal indicate that they require input from other tasks that have *not* been performed yet. This indicates an iteration in the process being modeled. One can draw boxes including the coupled tasks around the diagonal. If these boxes indicate the same tasks that the company recognizes as iterative, then the process already covers these coupled tasks. However, if the boxes indicate other *unaccounted* coupled tasks, this means the organization might need to rethink its process because there are iterations not properly supported. Unaccounted dependencies are represented by "O"s in the DSM matrix presented in
Figure 10.

Similarly to component-based DSMs, activity-based DSMS can be reorganized by rearranging the matrix to minimize iterations or of include tasks in the planned iterations. Other possibilities include having team members participating in coupled tasks, introducing new tasks in the process to simplify subsequent iterations, and redefining tasks within coupled groups (Eppinger 2001). Of course, the optimization of the DSM has to take into account domain knowledge about the process, which means that the new DSM has to be evaluated by managers in the organization. The new reorganized matrix, and development process, is presented in Figure 9.

Eppinger and his students have used DSM matrices to compare, improve and extend organizational processes of product development in different companies. For instance, Morelli, Eppinger and colleagues (1995) studied a project in a manufacturer of electrical technologies. Mainly, they wanted to find out if it is possible to predict technical communication in a project. In order to do this, they created a task-based DSM of the product development process by interviewing key informants. This task-based DSM was mapped into a team-based DSM, that is, instead of dealing with tasks of the development process, they used the teams performing the tasks. The idea is that by representing the dependencies between the tasks, this DSM also represents the communication paths that need to take place to handle these dependencies, therefore, this matrix can be used to predict technical communication in the project. Then, they collected weekly questionnaires with information regarding the communication frequency of members of the teams involved in the product development. After the aggregation of these questionnaires, they compared this resulting matrix of *actual* communication with the one of *predicted* communication. Among the results of the comparison, they concluded that a majority 81.1% of the interactions was predicted. More specifically, nearly all of the frequent and most of the occasional communications were predicted. These results suggest that team members dealing with interdependent tasks should not have communication barriers between them, since they need to interact to handle their dependencies. This can be used to allocate team members in distributed projects, for instance.

In more recent studies, Eppinger and his students combine different types of DSMs providing recommendations of how teams should be organized to minimize the coordination needs in product (and software) development projects. This will be discussed in section 6.3.

4.5 Social construction of interdependencies

In Karsten's (2003) work interdependencies are seen as "social practices constructed when people build mutual relationships between themselves". More specifically, she focuses on the interdependence construction and re-construction processes. This is very "in line" with Weick's work who suggests that rather than focusing on organizations, one should focus on *organizing*, an analytical shift from structure to process (Scott 2003, pg. 98). Karen's approach is based on the concept of collective structure, which exists when the behaviors of two or more persons become more inter-structured and repetitive (Weick 1979). In this process, "the others [interaction] become more predictable and familiar in their action that the others depend on." In short, interdependence in Karsten's work means a more or less *expected* contingent relationship that exists between two actors, therefore she focuses on the *process* of interdependence construction. This is important because it draws attention to the need of studying the "life-cycle" of a dependence or the set of phases that a dependence goes through.

Karsten uses Gidden's structuration theory (Giddens 1979) as her analytical framework. She draws from this theory four aspects of interdependence construction, namely, social-integration, time-space distantiation, institutionalization, and system integration. Then, she presents empirical data from three cases studies describing how these aspects take place. Karen also discusses how collaborative information technologies can slow down and, at the same time, help these four aspects.

4.6 Socialization of Software Development

In a study of process improvement using CMM models, Adler (2003) found out that process maturity drove the software development effort toward greater interdependence and, consequently, greater effort to manage these interdependencies through more collaborative forms of division of labor. The author studied four different units of a software development organization that had recently adopted process improvement programs. Two of them were certified as level-5 in the CMM model, while the other two as level-3. Using data collected through interviews and analyzed used activity theory, Adler found out that software developers had became more interdependent in their work, both horizontally and vertically, in contrast to the individualist and independent "hacker mode" of software production adopted before the process improvement program. Sometimes, these interdependencies between developers' tasks took a coercive form. However, interdependencies took a collaborative form more often because there was "extensive participation" of all software developers in the process improvement plan. Developers even reported seeing these interdependencies as one way of achieving a better product, a way of synchronizing the group effort because "the object of developers' work more stable and more intelligible". That is, two factors contributed to this collaborative aspect of interdependencies: (i) the reorganization of the work to "create" interdependencies; and (ii) the developers' awareness and understanding of the interdependencies, so that they could comprehend how each and every task contributes to the whole process.

4.7 Discussion

As mentioned before, one can observe that task interdependencies are often associated with a discussion about the coordination mechanisms required to deal with them. Furthermore, the degree of the interdependencies influences the coordination mechanisms such that the more interdependent two tasks are, the more difficult is to coordinate them. This is discussed by Thompson (1967) when he defines pooled, sequential and reciprocal interdependencies and their associated coordination mechanisms (standardization or rules, plans and schedules, and mutual adjustment) and by Adler (Adler 1995), who in addition, takes into account the temporal aspects of a project and how the coordination mechanisms evolve alongside the project.

Meanwhile, the DSM approach can be used to, more or less effectively, assess this degree of interdependence between tasks, that is this approach focus on the analysis of task dependencies to find out the degree of coordination effort required. Operations are applied to the DSM to reduce the dependencies between tasks and therefore facilitate their coordination. In short, the DSM approach focuses on the representation and analysis of tasks dependencies.

Karsten's and Adler's work take a different focus. Karsten is interested in understanding how dependencies get created, while Adler's results indicate one factor (software process improvement) that might lead to the "creation" of more interdependent tasks.

5 Multi-Models Approaches

The previous two sections described approaches that focused solely on either the product or the artifact model. In contrast, this section describes two approaches that address both models at the same time, namely process modeling languages and configuration management (CM) tools. Process modeling languages provide a better integration of the models, so that it is possible to identify product dependencies from the task dependencies. On the other hand, the integration is not so smooth in CM tools.

5.1 Process Modeling Languages

Software development processes are defined as "the coherent set of policies, organizations, structures, technologies, procedures, and artifacts that are needed to conceive, develop, and maintain a software product" (Fuggetta 2000). They describe the steps that need to be performed during a software development effort given the problem to be solved, the specific development project, and all particularities of the organization, environment and product being developed. At this point it is important to distinguish software process approaches from life-cycle descriptions such as the traditional waterfall and spiral models. These descriptions do not detail the building blocks of a process that are necessary for managing and coordinating projects (Fuggetta 2000).

Software processes can be used to many purposes, such as facilitating human communication and understanding of processes, learning and training newcomers, and automating process guidance and execution. However, in order to achieve these goals, it is necessary to be able to describe the software development processes being studied. Process modeling languages (PMLs) are were created to fulfill this role. Process descriptions provided by PMLs define relationships among the elements of the software process. For example, if a task must be performed before another, if an activity produces a particular output, or how artifacts relate to each other creating hierarchies of artifacts. By doing so, they describe dependencies in the software development process, that is, dependencies either among artifacts or tasks that need to be fulfilled during the enactment of the software process. That is, they describe how the several elements being modeled interact to achieve the ultimate goal of producing a software application. The insight that a software process, directly or indirectly, defines dependencies between software development artifacts, has been explored by development tools aiming to provide semi-automatic collection of traceability links (see section 3.5)

PMLs use basic modeling entities, such as artifacts, tasks, roles, and so on to describe software processes (Cugola and Ghezzi 1998). But, there is no agreement in the software process community about the set of entities. Indeed, several PMLs have been proposed in the literature and they can be classified according to their main paradigm[12]. For the purposes of the current discussion, I am interested in PMLs whose paradigm is either task-based or artifact-based PMLs. The difference between them is that in artifact-based PMLs the focus is on transformation of artifacts, while in task-based PMLS the focus is on the ordering of tasks. But, both are semantically equivalent (Barthelmess 2003). An example of a artifact-based PML is PROSYT (Cugola 1998). Process descriptions represented in PROSYT define relationships among the elements of the software process. However, these artifacts contain "operations" that can be applied to them. Preconditions and guards associated to artifacts and their "operations" indirectly describe the sequence of tasks to be performed and, consequently, the interdependencies between these tasks. Similarly, task-based PMLs describe software processes according to the tasks or activities that need to be performed to develop a software product. These PMLs are usually specified as a partial order of work tasks, based on ordering constraints (e.g., a task A must be executed before another task B) (Barthelmess 2003). However, they also include the set of artifacts being created by each task. Then, again, they define indirectly dependencies among the artifacts. Examples of these PMLs include SPADE and APP/A.

Workflow management systems are very similar to software process environments, but they are more generic and aim to model business processes. Most workflow management systems is task-based and also describe the artifacts being produced by these activities (Barthelmess 2003). An exception is the work of Lamarca and colleagues (1999), who support document-centered collaboration. This is achieved by associating active properties to the artifacts. These properties contain the knowledge about the dependencies between the tasks of the process being modeled.

5.2 Building Mechanisms in Configuration Management Tools

Configuration management (CM) is the process which controls the changes made to a system and manages the different versions of the evolving software product (Sommerville 2000). These systems provide several types of services, including versioning, system building, process support concurrent engineering, among others. In this section, I am interested in (i) *system building* because this service is the one that uses information about the dependencies between the products available in the CM system to (re)build the system, and (ii) *process support*, because it allows the specification of task dependencies. Both services are described below.

System building is "the process of combining the components of a system into a program which executes on a particular target configuration. This may involve compilation of some components and a linking process, which puts the object code together to make an executable system" (Sommerville 2000). This service facilitated the acceptance of CM systems among software developers because it was useful for them, in contrast to other services provided by a CM system that are more useful to the configuration manager (Estublier 2001).

Basically to (re)build a system, it is necessary to derive objects from other objects (source code). This is done taking into account build dependencies – rules that define how an object is derived from other objects. Building tools, like make and ant, automate the process of system building by using these rules to possibly minimize the number of required recompilations after a change is made. This is possible because such tools analyze the modification date of the source code: if it is later than the modification date of the object code with the same name, then

[12] The interested reader might refer to surveys of PMLs such as (Curtis, Kellner et al. 1992), (Ambriola, Conradi et al. 1997) and (Barthelmess 2003).

recompilation is necessary. This is the simplest mechanism used by CM tools, other more sophisticated can be used as well (Estublier, Leblang et al. 2005). This mechanism potentially reduces the rebuilding time from days to hours, or from hours to minutes. Make and ant store the building rules in text files, which are generated by CM systems based on the information about the dependencies they have available (Estublier 2001).

CM tools also provide process support (see section 5.1 for software processes). Some authors even argue that CM is one of the few software engineering domains in which process support has proven to be successful (Estublier 2001; Estublier, Leblang et al. 2005). In this case, the process support available in CM tools is limited to change control and trouble reporting. These are described using state transitions diagrams. Support for other parts of the development process, that is, *generic* process support, has not been widely accepted by users who found it too daunting and difficult (Estublier, Leblang et al. 2005).

6 Integrating the Product and Task Models

In separating product-based and task-based dependencies, it is possible to investigate different aspects of these phenomena. Because researchers were able to focus in one aspect at a time, it was possible to provide important contributions to these research areas. However, the results discussed in this section suggest that this same separation between task and product models provides only relatively narrow and clear-cut views of what can be understood as a broader phenomenon. Most importantly, these narrowed approaches also may remain insensitive to the complex and situated nature of the *relationships* among them. That is, as important as studying the product and task model, is studying their relationship: Does the product model influences and/or is influenced by the task model? If so, how? The studies presented in this section attempt to answer these questions by exploring how dependencies in one model influence the other. By doing that, it is possible to gain additional insights about interdependencies. Multi-models approaches, discussed in the previous section, do not address these issues because they only allow one to infer product dependencies from task dependencies. In other words, they explore both models at the same time instead of exploring how one affects the other.

6.1 Conway's Law

In 1968, Conway (1968) claimed that "organizations … are constrained to produce designs which are copies of the communication structures of these organizations." According to him, in any design process, several design options are "automatically" made *not* available to an organization because they do not reflect communication patterns of its members. Conway argues that the system structure will be stamped out with the communication structure of the organization because the communication needs of those doing the work are inevitably reflected in the system.

This relation – that has become known as Conway's Law – is also commonly stated as: "If you have four groups working in a compiler, you will get a four-pass compiler." This argument is more adequate to the initial version of a software system and it is based on the assumption that the organizational structure is immutable. Therefore, it describes a relationship from the task model to the product model. Moreover, Conway's argument should not be understood as a prediction, because organizations might change to facilitate the coordination of product development (see section 4.2). Instead, Conway should be read as an advice suggesting that software design can be facilitated *by matching the organizational structure and the product architecture*. Indeed, Conway's Law can be interpreted as an organizational pattern (Coplien and Harrison 2005, pg. 192):

> "If the parts of an organization (e.g., teams, departments, or subdivisions) do not closely reflect the essential parts of the product, or if the relationships between organizations do not reflect the relationships between product parts, then the project will be in trouble."

From this point of view, Conway's Law is seen as a bi-directional relationship between the product and the task model, that is, the product model influences the task model as much as the task model influences the product model. Misalignments between them should be avoided. Overall, Conway's argument acknowledges the importance of social (communication) aspects in software design. Based on this argument, he also pointed out directions for future research:

> "Even in a moderately small organization, it becomes necessary to restrict communication in order that people can get some 'work' done. Research which leads to techniques pointing more efficient communication among designers will play an extremely important role in the technology of system management."

In fact, four years later, Parnas proposed the principle of information hiding, which minimizes communication needs among software developers by restricting the information they exchange. Information hiding is discussed in the next section.

6.2 Parnas' Information Hiding

Modularity argues that systems should be organized in modules to facilitate their development, maintenance, and reuse. However, until 1972, there was no clear recommendation of how to divide a complex system into modules, which meant that they could still be highly interdependent of each other. Luckily, this was changed by Parnas when he proposed the principle of information hiding (Parnas 1972). According to this principle, software modules should only expose details that are not likely to change, therefore reducing the degree of interdependencies. If particular module's details are encapsulated within the module and do not affect other modules, they can be changed more easily without causing significant impact. Basically, Parna's proposes a distinction between module's interfaces and implementation. The interface is the stable part, while the implementation is the unstable one. Modules get connected to each other through their interfaces.

This principle motivates several mechanisms in programming languages that provide flexibility and protection from changes, including data encapsulation, interfaces, and polymorphism; and is one of the key principles behind object-oriented programming (Larman 2001).

In addition, it suggests a division of labor among the software engineers involved: Parnas defines a module as "a responsibility assignment rather than a subprogram", illustrating how a typical artifact-based approach (a module) interrelates with work tasks assignments. Or, how the product model influences the task model. This is clearly illustrated by Ghezzi and Jazayeri in their famous software engineering book:

> "If a design is composed of highly independent modules, it supports the requirements of large programs: Independent modules form the basis of work assignments to individual team members. The more independent the modules are, the more independently the team members can proceed in their work" (Ghezzi, Jazayeri et al. 2003, pg. 241).

Parna's information hiding has also been applied in product and organizational design. In these fields, it has been called only modularity. Accordingly, modular product design has been adopted by several industries including aircrafts, automobiles, consumer electronics, personal computers, among others (Sanchez and Mahoney 1996). The first modular computer, the IBM/360, was both a blessing and a curse for IBM. It increased sales and revenues, but created competition to IBM because it allowed other companies (mostly composed of former IBM employees) to produce better and/or cheaper IBM-compatible modules for memory, storage, printing, and so on. Baldwin and Clark (1997) argue that the modularity adopted by the computer industry is the *key* factor for its success.

Moreover, organizational science has also benefit from modularity because "the creation of modular product architectures not only creates flexible product design, but also enables the design of loosely coupled, flexible, 'modular' organization structures" (Sanchez and Mahoney 1996). This is only possible because well-defined interfaces between the products facilitate coordination practices, reducing the need for management and control over the module's associated personnel (Mintzberg 1979). Some authors even assert that modularity in design of products lead to- or at least ought to lead to- modularity in the design of organizations that produce such products (Langlois 1999).

6.3 Combined Design Structure Matrices

Steven Eppinger and his students at the MIT have conducted different studies to explore the relationship between product architecture and organizational structure, that is, how the architecture of a product (its components and associated dependencies) relate to the organizational structure (the division of labor in teams and their interactions). These studies are described in this section.

Sosa, Eppinger and colleagues (2002) conducted a study of a project that involved both software and hardware development in the telecommunications industry. They wanted to find out the influence of some factors in the frequency of technical communication. More specifically, the factors tested include: the importance of the interdependence between two systems in the product architecture; organizational bonds, distance, cultural and language differences between the parties dealing with these two systems; and the media (face to face, telephone or email) used by these parties for communication. They found out that communication frequency correlates positively with the importance of interdependence and organizational bonds, but decreases with distance. This result holds across all media studied, suggesting that "apparently, people involved in critically interdependent tasks or who share strong organizational bonds engage in a broad spectrum of communication means." Even when team members were non-collocated, higher communication frequencies were observed for highly interdependent pairs when compared to non-collocated independent pairs. These results reinforce the importance for managers to identify critical task dependencies in their organizations to facilitate intense communication among the team members involved in such interdependent tasks. In addition, the authors argue that by documenting communication frequencies, managers can uncover the underlying structure of products, or, more importantly, unidentified dependencies. To quote the authors: "tracking electronic-based communication frequencies can provide an easy and non-disruptive way to obtain the dependency structure of a development project."

In a different study, Sosa (2003) compared two different DSMs during the design of a commercial aircraft engine. It focused on the distinction between modular and integrative systems:

> "[A modular system is one where] design interfaces with other systems are clustered among a few physically adjacent systems, whereas integrative systems are those whose design interfaces span all or most of the systems that comprise the product due to their physically distributed or functionally integrative nature throughout the product."

The identification of these two types of systems is based on the analysis of the component-based DSM. The study focused on understanding teams' interactions: they wanted to compare the interaction of design teams that develop modular systems and the interaction of design teams that develop integrative systems.

Initially, they created a component-based DSM by collecting information from key informants regarding the several systems' interactions in the product. Then, they collected information about the frequency and importance of technical interactions among the teams in order to generate a team-based DSM. Finally, they compared both matrices aiming to identify matched and unmatched interactions[13]. Matched and unmatched interactions and design interfaces are defined in Figure 11.

Team Interactions	NO	**Unmatched design interfaces**	**Aligned absence** of interfaces and interactions
	YES	**Aligned presence** of interfaces and interactions	**Unmatched team interactions**
		YES	NO
		Design Interfaces	

Figure 11 - Mapping Design Interfaces and Team Interactions

When the component and team-based DSMs were compared, about 90% of the cells matched: they were cells where either known product dependencies that were matched by team interactions or cells with no product dependencies and no corresponding reported team interactions. This is another evidence that suggests a homomorphic relationship between the product model and the task model.

In addition, the authors "(…) found a statistically significant larger proportion of unpredicted team interactions associated with modular systems. Since … [these interactions] represent unrecognized design interfaces, we conclude that design interfaces across modular systems are more difficult for design experts to recognize than interfaces with integrative systems". I believe this result is somehow expected since modular systems are by definition "expected" to have more interactions internally than externally with other modules. Despite that, this suggests that special attention should be given to the "integrative" part of modular systems. In addition, as mentioned before, one can potentially identify undocumented dependencies based on documented team interactions. To make this process more cost effective, one could limit data collection of team interactions focusing on interactions across system boundaries, because those are more difficult to be recognized by experts.

Later, the unmatched cells in Figure 11 were examined by Sosa and colleagues in another study (Sosa, Eppinger et al. 2004). The results of this study indicate that unmatched design interfaces across modular systems occurred because "many strong cross-boundary design interfaces were perceived as weak interfaces and therefore no planning mechanisms were in place to address them." [emphasis in the original]. This suggests that "modularization itself may further hinder design teams' ability to handle interfaces across boundaries". On the other hand, unmatched team interactions across modular systems are "good" misalignments because they happened to solve unnoticed system-level dependencies.

This additional analysis of the same dataset used statistical network approaches to account for the "deviation from randomness embedded in [the] network data", that is, statistical methods assuming the independency of the cells can not be used because some cells (systems) are more likely to generate or attract linkages with other systems (Sosa, Eppinger et al. 2004).

Browning (2001), another Eppinger student, presented a classification of DSMs in four different types, previously described in section 3.6. In addition, he discusses how these models relate to each other. For the purposes of this paper, I am interested in only two types of matrices: component-based and activity-based. According to Browning's review of the literature, these models influence each other: the structure of the product influences its development process

[13] Note that this assumes a one to one relationship between product subsystems and teams, that is, for each subsystem there is only one team designing it.

(Baldwin and Clark 2000) (Nightingale 2000) and conversely, a legacy development process might constrain the structure of products, not allowing them to use architectures that would allow innovation.

6.4 Grinter's Recomposition

Grinter (2003) provides one of the few ethnographic accounts of dependencies in software development work[14]. In particular, Grinter studies the process of "recomposition", that is "the work necessary to ensure that a software product can be assembled from its component pieces". Recomposition is seen as the natural complement for decomposition, but while decomposition happens only once in the beginning of the project, recomposition occurs several times along the development process. Obviously, the process of recomposition needs to take into account the dependencies of a particular piece of software. Under this light, recomposition is then defined as "the work of coordinating and communicating enough to maintain a shared understand of the dependencies".

Based on ethnographic investigations in three different software development companies of varied sizes, Grinter documents a few individual and more often organizational "responses" to deal with the additional work of recomposing the software system. For instance, broadcasting emails was used in all organizations by small software development teams to support the management of dependencies because it helps developers to communicate their changes to those who depend upon the code being changed. This is a very basic method used by software developers, which obviously does not scale. Therefore, more formal mechanisms are used to manage dependencies. For example, the creation of build and release groups responsible, among other things, for ensuring that only the necessary changes are included in a particular build. Members of this team need to interact with all developers to make sure that all functionalities are included in the build.

Grinter's research illustrates how dependencies between software modules need to be properly managed by the software developers implementing or maintaining them. Furthermore, it illustrates how these technical dependencies require additional work to be managed pointing out to the influence of organizational factors in these dependencies: dependencies were established - or removed- between pieces of code because of organizational decisions. Finally, dependencies are seen as inevitable part of any software developer's work either as a desired goal (for instance when a developer can simplify his work by reusing other's developer code) or a problematic situation (whenever another's developer code breaks the build, therefore difficulting everybody else's work).

6.5 de Souza's Studies

de Souza and colleagues have conducted two different ethnographic field studies of software development teams focusing on how software developers manage dependencies in their work (de Souza, Redmiles et al. 2003; de Souza, Redmiles et al. 2004). Both are discussed in this section.

In the MVP study, de Souza and colleagues (2003) studied a software development team for eight weeks. Data collection methods included interviews and observations, while analysis was based on grounded theory. They identified a set of approaches used by the MVP team members to deal with interdependencies in their work. They were classified in formal (legitimately adopted by the organization) and informal approaches (emerging work practices adopted by the team).

[14] Other aspects of software development have been studied, such as distribution and collocation (Grinter, Herbsleb et al. 1999) (Herbsleb, Mockus et al. 2000) (Teasley, Covi et al. 2000), software design (Herbsleb, Klein et al. 1995), architectural design (Grinter 1999) (Smolander 2002), requirements engineering (Jirotka and Wallen 2000) and so on.

The formal approaches identified are: the software development process; the software development tools used (namely the configuration management and bug-tracking tools); and other approaches, such as the division of labor, formal meetings, and so on. The informal approaches are the adoption of conventions, "partial check-ins", "holding onto check-in's", "speeding up" the process, problem reports (PRs) that cross work boundaries, and the role of e-mail as a coordination mechanism.

In contrast to the organizational solutions studied by Grinter (see section 6.4), this study focused on the daily practices adopted by software developers. By focusing on them, it is possible to identify how developers perceive their interdependencies and how, in their work, they act and react to them. The results suggest that formal and informal practices are used to carefully integrate new changes in the code repository and to protect one's work from the effect of others' changes, that is, to handle the transition from private to public work and vice-versa (de Souza, Redmiles et al. 2003). According to the model proposed in section 2.2, it is possible to conclude that product dependencies create task dependencies, in other words, dependencies in the software create dependencies between the tasks to be performed.

In a second field study, de Souza and colleagues (2004) adopted a different approach, they focused on the management of interdependencies among teams of developers. In this case, application programming interfaces programming interfaces (APIs) played a fundamental role. Among professional software engineers the term API is coming to mean any well-defined interface that defines the service that one component, module, or application provides to other software elements. APIs are largely adopted by industry because they support the separation of interface from implementation (Fowler 2002). APIs are one among several instantiations of the information hiding principle (see section 6.2).

Overall, the findings of this field study show that APIs simultaneously allow and constrain collaboration among software developers. On the one hand, they facilitate collaboration during the decomposition process because they establish a shared understanding among developers of what needs to be done and at some level formalize this agreement. They also reinforce the organizational boundaries of team membership. On the other hand, because they reinforce team memberships, APIs limit collaboration among software developers from *different* teams. Since these developers belong to different teams, one in each side of the API, communication and coordination among them is more difficult due to the organizational boundaries (Mintzberg 1979). In short, these APIs reify organizational boundaries dividing the work necessary to develop software, but at the same time create an isolation that as a side effect reduced team awareness about other teams' work (de Souza, Redmiles et al. 2004).

To summarize, APIs are an example of a *technical* approach used by teams of developers to deal with the interdependencies in their work providing another evidence of the alignment that is necessary between the product and task models. Some of the problems identified might be explained because of the misalignment. The role of APIs suggested the authors that the source-code itself could be used to identify the social dependencies, that is, the coordination and communication needs among software developers. Indeed, these authors explored this idea and even developed tools to address this aspect. This is discussed below.

Ariadne (de Souza, Dourish et al. 2004; de Souza, Redmiles et al. 2004) is a Java-based tool that extracts dependencies from Java source-code and "translates" them in sociograms, therefore describing the dependencies between the developers responsible for the code. Using the framework described in this paper, Ariadne translates product dependencies into actor dependencies. Augur (Froehlich and Dourish 2004) is a system for visualizing software systems in which properties of the software system are mapped to color and other features of a graphical display of the source code itself. The initial system was extended to support interdependence analysis and used to understand the practice of open-source software development (de Souza, Froehlich et al. 2005).

6.6 Discussion

Relationships between the product and task models were recognized over 30 years ago. For instance, Conway's Law illustrates how the communication patterns in a software development organization and the software itself are homomorphic as consequence of the communication needs of the people doing the work in the integrated parts. Similarly, Parnas defines a module as "a responsibility assignment rather than a subprogram", illustrating how a typical artifact-based approach (a module) interrelates with work tasks assignments.

Parnas' and Conway's work are not empirical, however. More recently, a series of studies in both software development (Sosa, Eppinger et al. 2002; Grinter 2003; de Souza, Redmiles et al. 2004) and product development (Sosa, Eppinger et al. 2003) started to provide evidence supporting their arguments. In addition, (Crowston 2003) briefly comments the influence of the task model on the product model:

> "One manager interviewed suggested that in software development, a manager might divide a system into modules based on the abilities of the programmers who will work on them, rather than on some *a priori* division of the problem."

Figure 12 below describes the relationship between the different models of the framework, and where the results of each theoretical and empirical work are located.

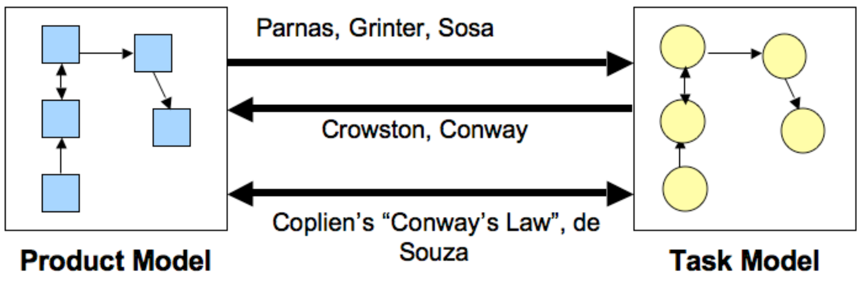


Figure 12 - Relationships between models

A more recent approach has been to create or adapt tools (Ariadne and Augur) to explore the relationships between these models. This has been applied to software projects, because programming languages have a defined syntax and semantics that can be analyzed for the identification of interdependencies. Program analysis techniques are combined with visualization approaches to present information about the web of dependencies (technical and social) in which software developers are embedded.

7 Common Themes

The framework and models used in this paper are used to properly account the different perspectives to the study of dependencies. In addition, they enable one to identify the common themes that arise in the different models. The most common of these themes is the *degree* or strength of the dependence relationship. For instance, in the product model it is possible to identify the concept of *coupling* that is used as a measure of the interdependencies between two software modules (Stevens, Myers et al. 1974). A scale from less desirable to most desirable coupling is even defined (Constantine and Yourdon 1979). Similarly, in the task model, Thompson (1967) defines three different types of dependencies (reciprocal, sequential and pooled) and discusses how they form a scale:

“These three levels of interdependencies form a Guttman-type scale, in that elements or processes that are reciprocally interdependent also exhibit sequential and pooled interdependence, and processes that are sequentially interdependent also exhibit pooled interdependence.”(Scott 2003)

Often, this measure of the degree of dependence is used as an indicator of the value of a dependence, that is, if the dependence relationship is desired or not. A positive dependence can be identified in approaches based on the product model, such as software engineering traceability approaches that suggest that dependencies between software artifacts should be represented, recorded, and maintained to properly support software evolution, reuse, testing, and so on (Spanoudakis and Zisman 2004). Similarly, high degrees of cohesion are desired among the components of a software module. On the other hand, in the same product model, dependencies can express an undesired relationship. For instance, Pimmler and Eppinger (1994) report a study of a large commercial aircraft engine where engineers were interviewed in order to classify dependency as desired or undesired. An undesired dependency is the transfer of energy from one particular component to another.

In the task model, researchers in management science also indicate the need for the distinction between desired and undesired dependencies. Sorenson (2003), for example, discusses how interdependence among tasks in an organizational process hinders organizational learning. This happens because these interdependencies make extremely difficult the isolation of tasks that are the source of poor performance or the isolated changes in the process that result in process improvement. Even when it is possible to isolate such tasks, the interdependence with other tasks makes changes very difficult.

In addition to being able to measure the degree of dependencies between entities, another common theme is the issue of representing them, that is, how the information about the existing set of dependencies is collected, represented, manipulated and presented. In contrast to most product model approaches that only use graph representations such as the program dependence graph (Ferrante, Ottenstein et al. 1987), task model approaches use several representations including matrices (Steward 1981; Eppinger 2001) and graphs for workflow representations (Nutt 1996). Despite these different approaches, it is not difficult to observe that they are equivalent, that is, matrices can be represented as graphs and vice-versa.

In short, in my analysis of the different approaches studied in each model, I identified three common themes:

1. Value: are dependencies good or bad relationships?
2. Degree: how weak or strong is the dependence relationship between the two entities?
3. Representation: how dependencies are represented so that they can be understood, visualized, managed, and analyzed?

8 Promising Research Topics

In the previous section, I have discussed common aspects found in both the task and product models. Another commonality between these approaches is the study of interdependencies as *static* entities. In general, the product model focuses in dependencies among products, but does not take into account *how* these dependencies came to be. Similarly, the task model studies dependencies between tasks and their effect to these tasks’ coordination, but not how tasks get more or less interdependent. With the exception of Karsten’s (2003) work in the social construction of interdependencies in the task model, no other study focuses on temporal aspects. However, her work has limitations because of her narrow definition of interdependencies and her focus on the role of collaborative information technologies in this construction process (see section 4.5). This suggests that a potential area for research is the study of the *evolution* of dependencies. The idea is try to answer questions such as: how, when and why dependencies get created or cease to exist? Which work is involved in managing this evolutionary process? And, finally, can tools help in this process? This is particularly important when one wants to reuse modules. That suggests the question of how new entities are integrated into a web of existing interdependent entities? In short, the idea is to understand the temporal aspects of dependencies, which would allow one to understand how they evolve. One approach can be to study the evolution of dependencies through DSMs (Eppinger, personal communication, March 2005).

Additionally, section 6 discusses the relationship between the task and product models. This relationship indicates that the study of these models' evolution should not be done independently, because the evolution of any of these models potentially causes the other to evolve as well. Understanding the evolution of interdependencies in isolated models is as important as understanding their *co-evolution*.

Empirical work has been done to study dependencies in each model. For instance, (Dahlstedt and Persson 2003) and (Ramesh and Jarke 2001) have studied requirements interdependencies in software traceability models. Similarly, in the task model, researchers have "validated" Thompson's predictions about the degree of interdependence and the coordination mechanisms used to handle them (e.g., Van de Ven, Delbecq et al. 1976). Empirical evidence also exists indicating the relationship between the task and product models (Morelli, Eppinger et al. 1995; Sosa, Eppinger et al. 2002; de Souza, Redmiles et al. 2003; Grinter 2003; Sosa, Eppinger et al. 2003; de Souza, Redmiles et al. 2004). However, only a small part of this evidence ((de Souza, Redmiles et al. 2003; Grinter 2003; de Souza, Redmiles et al. 2004)) answers the following questions: how social actors deal with interdependencies in their work? What kind of tools they use? How they use these tools? What are the ways in which these actors deal with interdependencies? What problems emerge when these interdependencies are not properly handled? These partial results are not enough given the importance of the management of interdependencies in cooperative work. I believe more empirical studies of the practices by which social actors deal with their interdependencies are necessary, in particular in software development projects.

Likewise, most end-user tools that aim to support the management of interdependencies focus in one model – architectural dependence analysis tools, for instance[15]. New tools aiming to bridge the two models are important and might prove useful. While DSMs are a very compact representation, they do not scale for software projects with hundreds of thousand of methods. Of course, software systems can and indeed are aggregated into several levels of abstractions (e.g., from methods to classes and from classes to packages in Java). Because software systems are organized hierarchically, structures such as Treemaps and alike are a potential visualization technique to be used (Wattenberg, personal communication, July 2004). But research is still needed to identify the "right" level of abstraction for managers and end-users. That is, visualization techniques are necessary, but to whom? Who are the users interested in understanding and visualizing their dependencies? de Souza's studies in the MVP project highlight how software developers in a group are aware of these interdependencies and take action to manage them. Conversely, the MCW study indicates how the lack of awareness about one's interdependencies might create problems. This suggests that a potential user interested in visualizing interdependencies are the software engineers themselves, however, this still needs to be investigated.

[15] Note that design structure matrices provide a unique representation of these (and other models), however they are not aimed for end-users. Instead, they are meant for managers trying to understand the overall product development process.

9 Conclusions and Future Work

Modularity is a principle for managing complexity that has been applied to diverse human enterprises, including, but not limited to organizations, products, and development processes. A consequence of the decomposition needed for modularization is the need for those modules to interact or collaborate to compose a larger complex system. For instance, software modules exchange data and control to perform complex computations, an automobile motor exchanges energy with the automobile electrical system, so that it can provide energy to turn on the radio or the air-conditioner. These interactions, also called interdependencies, need to be handled (i.e., analyzed, engineered, documented, developed, and managed) efficiently and effectively if the complex system is to be built in a robust and successful manner. Different disciplines have already recognized this need to handle dependencies and each one has developed approaches according to the complex system focus of their research. This paper survey these approaches in the areas of software engineering, product development, management science, computer-supported cooperative work and organization science by placing them into a unique framework. Hopefully, this framework creates a common vocabulary for researchers and summarizes the body of knowledge in the study of dependencies. By using this framework, it is possible to compare approaches identifying common themes among them and suggest directions for future research. I hope to have motivated these directions, but much analytical and empirical work remains.

This survey did not explore the organizational model. Instead, it assumed that organizational dependencies could be studied as either task or product dependencies. This might be true in some cases (e.g., resource-dependence theory), but I can not argue that it is true in all cases. By studying other approaches that focus on interdependencies between organizations one might find new common themes or be able to map approaches from one model others. Additional work is necessary.

Acknowledgements

This research was supported by the U.S. National Science Foundation under grant numbers 0205724 and 0326105, IBM, and by the Brazilian Government under CAPES grant BEX 1312/99-5.

10 References

Adler, P. S. (1995). "Interdepartmental Interdependence and Coordination: The Case of the Design/Manufacturing Interface." Organization Science **6**(2): 147-167.

Adler, P. S. (2003). Practice and Process: The Socialization of Software Development. Los Angeles, CA, Marshall School of Business, University of Southern California.

Aho, A. V., R. Sethi, et al. (1986). Compilers: Principles, Techniques and Tools, Addison-Wesley.

Allen, R. and D. Garlan (1994). Formalizing Architectural Connection. Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, IEEE Press.

Ambriola, V., R. Conradi, et al. (1997). "Assessing Process-Centered Software Engineering Environments." ACM Transactions on Software Engineering and Methodology **6**(3): 283-328.

Baldwin, C. Y. and K. B. Clark (1997). "Managing in an age of modularity." Harvard Business Review **75**(5): 84-92.

Baldwin, C. Y. and K. B. Clark (2000). Design Rules, Vol. 1: The Power of Modularity. Cambridge, MA, The MIT Press.

Barthelmess, P. (2003). "Collaboration and Coordination in process-centered development environments: A review of the literature." Information and Software Technology. **45**(13): 911-928.

Brown, A. W. and K. C. Wallnau (1998). "The Current State of CBSE." IEEE Software **15**(5): 37-46.

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.

Callahan, D., A. Carle, et al. (1990). "Constructing the Procedure Call Multigraph." IEEE Transactions on Software Engineering **16**(4): 483-487.

Carlshamre, P., K. Sandahl, et al. (2001). An industrial survey of requirements interdependencies in software product release planning. International Symposium on Requirements Engineering, Toronto, Canada.

Constantine, L. L. and E. Yourdon (1979). Structure design: fundamentals of a discipline of computer program and systems design. Englewood Cliffs, N.J., Prentice Hall.

Conway, M. E. (1968). "How Do Committees invent?" Datamation **14**(4): 28-31.

Coplien, J. O. and N. B. Harrison (2005). Organizational Patterns of Agile Software Development. Upper Sadle River, NJ, Pearson Prentice Hall.

Crowston, K. (2003). A taxonomy of organizational dependencies and coordination mechanisms. Tools for Organizing Business Knowledge: The MIT Process Handbook. T. W. Malone, K. Crownston and G. Herman. Cambridge, MA, MIT Press.

Cugola, G. (1998). "Tolerating Deviations in Process Support Systems via Flexible Enactment of Process Models." IEEE Transactions on Software Engineering **24**(11): 982-1001.

Cugola, G. and C. Ghezzi (1998). "Software Processes: a Retrospective and a Path to the Future." Software Process - Improvement and Practice **4**(3): 101-123.

Curtis, B., M. I. Kellner, et al. (1992). "Process Modeling." Communications of the ACM **35**(9): 75-90.

Dahlstedt, A. G. and A. Persson (2003). Requirements Interdependencies - Molding the State of Research into a Research Agenda. The 9th International Workshop on Requirements Engineering: Foundation for Software Quality - REFSQ'03, held in conjunction with CAiSE, Velden, Austria.

de Souza, C. R. B., P. Dourish, et al. (2004). From Technical Dependencies to Social Dependencies. Workshop on Social Networks for Design and Analysis: Using Network Information in CSCW, Chicago, IL, USA.

de Souza, C. R. B., J. Froehlich, et al. (2005). Seeking the Source: Software Source Code as a Social and Technical Artifact (submitted). European Conference on Computer Supported Cooperative Work, Paris, France.

de Souza, C. R. B., D. Redmiles, et al. (2004). How a Good Software Practice thwarts Collaboration - The Multiple roles of APIs in Software Development. Foundations of Software Engineering, Newport Beach, CA, USA, ACM Press.

de Souza, C. R. B., D. Redmiles, et al. (2004). Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. Conference on Computer-Supported Cooperative Work (CSCW '04), Chicago, IL, USA, ACM Press.

de Souza, C. R. B., D. Redmiles, et al. (2003). Management of Interdependencies in Collaborative Software Development: A Field Study. International Symposium on Empirical Software Engineering (ISESE'2003), Rome, Italy, IEEE Press.

Eppinger, S. D. (2001). "Innovation at the Speed of Information." *Harvard Business Review*(January): 3-11.

Estublier, J. (2001). Software Configuration Management: A Roadmap. Future of Software Engineering, Limerick, Ireland, ACM Press.

Estublier, J., D. Leblang, et al. (2005). "Impact of Software Engineering Research on the Practice of Software Configuration Management (to appear)." ACM Transactions on Software Engineering and Methodology.

Fenton, N. E. and S. L. Pfleeger (1997). Software Metrics: A Rigorous & Practical Approach. London, PWS Publishing Company.

Ferrante, J., K. J. Ottenstein, et al. (1987). "The program dependence graph and its use in optimization." ACM Transactions on Programming Languages and Systems (TOPLAS) **9**(3): 319-349.

Fowler, M. (2002). "Public versus Published Interfaces." IEEE Software **19**(2): 18-19.

Froehlich, J. and P. Dourish (2004). Unifying Artifacts and Activities in a Visual Tool for Distributed Software Development Teams. International Conference on Software Engineering, Edinburgh, UK.

Fuggetta, A. (2000). Software Processes: A Roadmap. Future of Software Engineering, Limerick, Ireland.

Garlan, D., R. T. Monroe, et al. (1995). ACME: An Architectural Interconnection Language. Pittsburgh, PA, Carnegie Mellon University.

Ghezzi, C., M. Jazayeri, et al. (2003). Fundamentals of Software Engineering, Prentice Hall.

Giddens, A. (1979). Central Problems in Social Theory: Action, Structure and Contradiction in Social Analysis. London, UK, Macmillan.

Grinter, R., J. Herbsleb, et al. (1999). The Geography of Coordination: Dealing with Distance in R&D Work. ACM Conference on Supporting Group Work (GROUP '99), Phoenix, AZ, ACM Press.

Grinter, R. E. (1999). System Architecture: Product Designing and Social Engineering. Work Activities Coordination and Collaboration, San Francisco, CA, USA, ACM Press.

Grinter, R. E. (2003). "Recomposition: Coordinating a Web of Software Dependencies." Journal of Computer Supported Cooperative Work **12**(3): 297-327.

Hatcliff, J., X. Deng, et al. (2003). Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. International Conference on Software Engineering, Portland, Oregon.

Herbsleb, J., A. Mockus, et al. (2000). Distance, Dependencies, and Delay in a Global Collaboration. ACM Conference on Computer-Supported Cooperative Work (CSCW 2000), Philadelphia, PA, ACM Press.

Herbsleb, J. D., H. Klein, et al. (1995). "Object-Oriented Analysis and Design in Software Project Teams." Human-Computer Interaction **V10**(N2-3): 249-292.

Horwitz, S. and T. Reps (1992). The use of program dependence graphs in software engineering. International Conference on Software Engineering, Melbourne, Australia.

Jirotka, M. and L. Wallen (2000). Analyzing the workplace and user requirements: challenges for the development of methods for requirements engineering (chapter 12). Workplace Studies. P. Luff, J. Hindmarsh and C. Heath. Cambridge, UK, Cambridge University Press**:** 242-251.

Karsten, H. (2003). "Constructing Interdependencies with Collaborative Information Technology." Computer Supported Cooperative Work **12**(4): 437-464.

Lakhotia, A. (1993). Constructing call multigraphs using dependence graphs. 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, ACM Press.

Lamarca, A., W. K. Edwards, et al. (1999). Taking the Work out of Workflow: Mechanisms for Document-Centered Collaboration. European Conference on Computer-Supported Cooperative Work, Copenhaguem, Denmark.

Langlois, R. N. (1999). Modularity in Technology, Organization, and Society. Department of Economics Working Paper Series. Storrs, CT, Department of Economics, University of Connecticut.

Larman, G. (2001). "Protected Variation: The Importance of Being Closed." IEEE Software **18**(3): 89-91.

Lindvall, M. and K. Sandahl (1996). "Practical Implications of Traceability." Software Practice and Experience **26**(10): 1161-1180.

Luckham, D. C., J. J. Kenney, et al. (1995). "Specification and Analysis of System Architecture Using Rapide." IEEE Transactions on Software Engineering **21**(4): 336ˆ355.

Malone, T. W. and K. Crowston (1994). "The Interdisciplinary Study of Coordination." ACM Computing Surveys **26**(1): 87-119.

McIlroy, M. D. (1968). Mass-Produced Software Components. NATO Science Committee, Garmisch, Germany.

Mintzberg, H. (1979). The Structuring of Organizations: A synthesis of the research. Englewood Cliffs, NJ, Prentice-Hall.

Morelli, M. D., S. D. Eppinger, et al. (1995). "Predicting Technical Communication in Product Development Organizations." IEEE Transactions on Engineering Management **42**(3): 215-222.

Mortensen, M. and P. Hinds (2002). Fuzzy Teams: Boundary Disagreement in Distributed and Collocated Teams. Distributed Work: New Research on Working across Distance Using Technology. P. Hinds and S. Kiesler, MIT Press**:** 283-308.

Murphy, G., D. Notkin, et al. (1998). "An Empirical Study of Static Call Graph Extractors." ACM Transactions on Software Engineering and Methodology **7**(2): 158-191.

Murphy, G., D. Notkin, et al. (1995). Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, New York, NY, USA, ACM Press.

Nightingale, P. (2000). "The product-process-organization relationship in complex development projects." Research Policy **29**: 913-930.

Nutt, G. J. (1996). "The evolution toward flexible workflow systems." Distributed Systems Engineering(3): 276-294.

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

Pimmler, T. U. and S. D. Eppinger (1994). Integration Analysis of Product Decompositions. ASME Sixth International Conference on Design Theory and Methodology, Minneapolis, MN, USA.

Podgurski, A. and L. A. Clarke (1989). The Implications of Program Dependencies for Software Testing, Debugging, and Maintenance. Symposium on Software Testing, Analysis, and Verification.

Ramesh, B. and M. Jarke (2001). "Towards Reference Models for Requirements Traceability." IEEE Transactions on Software Engineering **27**(1): 58-93.

Sanchez, R. and J. T. Mahoney (1996). "Modularity, Flexibility, and Knowledge Management in Product and Organization Design." Strategic Management Journal **17**(Winter Special Issue): 63-76.

Scott, W. R. (2003). Organizations: Rational, Natural, and Open Systems. Upper Saddle River, NJ, Prentice Hall.

Shrivastava, S. K. and S. M. Wheater (1999). "Workflow management systems - Guest Editor's Introduction." IEEE Concurrency **7**(3): 16-17.

Simon, H. A. (1996). The Architecture of Complexity: Hierarchical Systems. The Sciences of the Artificial. Cambridge, MA, The MIT Press**:** 183-216.

Smolander, K. (2002). Four metaphors of architecture in software organizations: finding out the meaning of architecture in practice. In Proceedings of the First International Symposium in Empirical Software Engineering, Nara, Japan, IEEE Press.

Sommerville, I. (2000). Software Engineering, Addison-Wesley Publishing Co.

Sorenson, O. (2003). "Interdependence and Adaptability: Organizational Learning and the Long-Term Effect on Integration." Management Science **49**(4): 446-463.

Sosa, M. E., S. D. Eppinger, et al. (2002). "Factors that influence Technical Communication in Distributed Product Development: An Empirical Study in the Telecommunications Industry." IEEE Transactions on Engineering Management **49**(1): 45-58.

Sosa, M. E., S. D. Eppinger, et al. (2003). "Identifying Modular and Integrative Systems and Their Impact on Design Team Interactions." ASME Journal of Mechanical Design **125**: 240-252.

Sosa, M. E., S. D. Eppinger, et al. (2004). "The Misalignment of Product Architecture and Organizational Structure in Complex Product Development." Management Science **50**(12): 1674-1689.

Spanoudakis, G. and A. Zisman (2004). Software Traceability: A Roadmap. Handbook of Software Engineering and Knowledge Engineering. S. K. Chang, World Scientific Publishing Co.

Stafford, J. A. and A. L. Wolf (2001). "Architecture-Level Dependence Analysis for Software Systems." International Journal of Software Engineering and Knowledge Engineering **11**(4): 431-453.

Stafford, J. A., A. L. Wolf, et al. (1998). Architecture-Level Dependence Analysis for Software Systems. International Workshop on the Role of Software Architecure in Testing and Analysis (ROSATEA), Marsala, Sicily, Italy.

Staudenmayer, N. A. (1997). Interdependency: Conceptual, Empirical, & Practical Issues. Cambridge, MA, Sloan School of Management, Massachusetts Institute of Technology.

Stevens, W. P., G. J. Myers, et al. (1974). "Structured Design." IBM Systems Journal **13**(2): 115-139.

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.

Sullivan, K. J., W. G. Griswold, et al. (2001). The Structure and Value of Modularity in Software Design. Joint Proceedings of the European Software Engineering/ ACM SIGSOFT Foundations of Software Engineering Conference, Vienna, Austria.

Szyperski, C. (1998). Component Software - Beyond Object-Oriented Programming, Addison-Wesley / ACM Press.

Taylor, R. N., N. Medvidovic, et al. (1996). "A Component- and Message-Based Architectural Style for GUI Software." IEEE Transactions on Software Engineering **22**(6): 390-406.

Teasley, S., L. Covi, et al. (2000). How Does Radical Collocation Help a Team Succeed? Conference on Computer Supported Cooperative Work, Philadelphia, PA, USA, ACM Press.

Thompson, J. D. (1967). Organizations in Action: Social Sciences of Administrative Theory. New Brunswick, Transaction Publishers.

Van de Ven, A. H., A. L. Delbecq, et al. (1976). "Determinants of Coordination Modes within Organizations." American Sociological Review **41**(2): 322-338.

Vieira, M. R. E. (2003). A Compositional Approach for Analyzing Dependencies in Component-Based Systems. Information and Computer Science. Irvine, University of California, Irvine. Doctor of Philosophy: 209.

Vieira, M. R. E., M. Dias, et al. (2000). Analyzing Software Architectures with Argus-I. Research Demonstration at the International Conference on Software Engineering, Limerick, Ireland, IEEE Press.

Vieira, M. R. E. and D. J. Richardson (2002). The Role of Dependencies in Component-Based System Evolution. International Workshop on Principles of Software Evolution, Orlando, Florida.

Weick, K. E. (1979). The Social Psychology of Organizing, McGraw-Hill Humanities.

Weiser, M. (1984). "Program Slicing." IEEE Transactions on Software Engineering **SE-10**(4): 352ˆ357.

Wilde, N. (1990). Understanding Program Dependencies. Pittsburgh, PA, Software Engineering Institute - Carnegie Mellon University.

Yassine, A. A. (2004). "An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure MAtrix (DSM) Method." Quaderni di Management(9).

Zhao, J. (1997). Using Dependence Analysis to Support Software Architecture Understanding. New Technologies on Computer Software. M. Li, International Academic Publishers**:** 135-142.