Sometimes you need to see through walls: A field study of application programming interfaces

Conference Paper · January 2004

DOI: 10.1145/1031607.1031620 · Source: DBLP

5 authors, including:

Cleidson De Souza
VALE
171 PUBLICATIONS **1,673** CITATIONS

SEE PROFILE

David Redmiles
University of California, Irvine
194 PUBLICATIONS **3,681** CITATIONS

SEE PROFILE

David Millen
IBM
128 PUBLICATIONS **4,233** CITATIONS

SEE PROFILE

Sometimes You Need to See Through Walls — A Field Study of Application Programming Interfaces

Cleidson R. B. de Souza[1,2]
[1]Universidade Federal do Pará
Departmento de Informática
Belém, PA, Brasil
55-91-211-1405
cdesouza@ics.uci.edu

David Redmiles[2]
[2]University of California, Irvine
Department of Informatics
Irvine, CA, USA
1-949-824-3823
redmiles@ics.uci.edu

Li-Te Cheng[3] David Millen[3] John Patterson[3]
[3]IBM T. J. Watson Research Center
Collaborative User Experience Group
Cambridge, MA, USA
1-617-577-8500
{li-te_cheng, david_r_millen, john_patterson}@us.ibm.com

ABSTRACT
Information hiding is one of the most important and influential principles in software engineering. It prescribes that software modules hide implementation details from other modules in order to decrease the dependency between them. This separation also decreases the dependency among software developers implementing modules, thus simplifying some aspects of collaboration. A common instantiation of this principle is in the form of application programming interfaces (APIs). We performed a field study of the use of APIs and observed that they served many roles. We observed that APIs were successful indeed in supporting collaboration by serving as contracts among stakeholders as well as by reifying organizational boundaries. However, the separation that they accomplished also hindered other forms of collaboration, particularly among members of different teams. Therefore, we think argue that API's do not only have beneficial purposes. Based on our results, we discuss implications for collaborative software development tools.

Categories and Subject Descriptors
D.2.9 [**Management**]: D.2.11 [**Software Architectures**]: H.4.1 [**Office Automation**]: Groupware; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms: Human Factors

Keywords: Interfaces, application programming interfaces, collaborative software development, qualitative studies.

1. INTRODUCTION
Walls serve many useful purposes within a building. They create private and quiet spaces, demark function, support special tasks, protect against the environment, etc. In a work setting, an individual office can facilitate certain kinds of solitary work just as conference rooms can facilitate certain kinds of collaborative work. But the same walls that create these useful spaces need doors for people to enter and exit, as well as windows to connect environments and support visual awareness [13]. In collaborative software development, *application programming interfaces* (APIs) act as walls facilitating many useful purposes. However, in a field study we conducted on their use in software development, we learned that the wall-like structure that APIs when interact with some particular organizational settings create can lead to some difficulties in collaboration. In terms of our metaphor, the walls of APIs also require doors and widows. In particular, doors and windows are needed to avoid breakdowns in communication and coordination attributable to certain kinds of interdependencies in collaborative software development [7] [20].

The field of software engineering has recognized some technical problems created by interdependencies and developed tools, approaches, and principles to deal with them. Configuration management and issue-tracking systems are examples of such tools, and the adoption of software development processes [8, 9] exemplifies an organizational approach. One of the most important and influential principles used to manage dependencies is the idea of information hiding proposed by Parnas [30]. According to this principle, software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" [24]. Information hiding aims to provide a principle that guides the decomposition of a software system into pieces (called modules) that decreases the dependency (or coupling) between any two modules. By following this principle, changes to one module do not severely impact other modules. In addition to suggesting a division of labor among the software engineers involved, this principle motivates several mechanisms in programming languages that provide flexibility and protection from changes, including data encapsulation, interfaces, and polymorphism [24]. In particular, separating interface specifications from their implementation is a growing trend in software design [14]. Furthermore, interface specifications are also helpful in the coordination of developers working with different components:

> *Interface specifications play the **well-known** role of helping to coordinate the work between developers of different components. If the designers of two components agree on the interface, then design of the internals of each component can go forward relatively independently. Designers of component A need not know much about the design decisions made about component B, so long as both sides honor their well-*

CSCW'04, November 6–10, 2004, Chicago, Illinois, USA.

specified commitments about how the two will hook together. [15] [emphasis added]

Application programming interfaces (APIs) are a fairly common example of interfaces supported by the underlying programming language that allow one software component to access programmatically another component [10]. They are commonly used in the industry to divide collaborative and distributed software development work, and are regarded as "the only scalable way to build systems from semi-independent components" [15].

This paper describes a field study of an organization's usage of APIs to understand how they facilitate the management of interdependencies and therefore facilitate collaborative software development. Our analysis found out that APIs support collaboration by serving as contracts among stakeholders and by reifying organizational boundaries. The separation that APIs accomplished, however, hindered other forms of collaboration, particularly among members of different teams. Therefore, we think it is noteworthy to know that API's do not only have beneficial purposes.

The rest of the paper is organized as follows. The next section describes the concepts behind APIs and explains their adoption by industry. The reader familiar with this concept might skip this section. Section 3 then presents the research site and methods used in our study. Sections 4, 5, and 6 describe our findings of how the organization and teams we studied go about developing, using, and maintaining APIs. Mundane and expected observations about collaboration using APIs are interpreted as "advantages," while exceptional uses or problems are interpreted as disadvantages. Sections 7 and 8 respectively discuss the data collected and the implications of our findings to the design of collaborative environments, especially for software development. Finally, conclusions and ideas for future work are presented.

2. APPLICATION PROGRAMMING INTERFACES

An API is a well-defined interface, usually supported by the underlying programming language, that allows one software one software component to access programmatically another component [10]. A more "formal" definition provided by the Software Engineering Institute is the following [1]:

> *Application Programming Interface (API) is an older technology that facilitates exchanging messages or data between two or more different software applications. API is the virtual interface between two interworking software functions, such as a word processor and a spreadsheet. (…) An API is the software that is used to support system-level integration of multiple commercial-off-the-shelf (COTS) software products or newly-developed software into existing or new applications.*

Although the definition above presents APIs as interfaces between software applications, 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 others software elements. In the rest of the text, we will use the terms component, module and software applications indistinctly, since they do not change the purpose of using APIs.

The word "interface" in the abbreviation is used to explicitly indicate that APIs are constructs that exist in the boundaries of at least two different software applications. For instance, the Microsoft Windows API allows a program to access and use resources of the underlying operating system such as file system, scheduling of processes, and so on. Typically, in a programming language such as Java, an API corresponds to a set of public methods of classes and interfaces, and the associated documentation (in this case, javadoc files). The two (or more) applications that an API divides are often developed by different teams, and hardly ever individuals.

APIs are largely adopted by industry because they support the separation of interface from implementation, a growing trend in software design [14]. The main advantage of this approach is the possibility of separating modules into public (the API itself) and private (the implementation of the API) parts so changes to the private part can be performed without impacting the public one and therefore minimizing the dependencies between these two parts.

In the rest of the text, we adopt the terms *API consumers* and *API producers*. *API consumers* are software developers who write code with method calls to an API, and *API producers* are software developers who write the API implementation.

An important aspect of any API is *stability*. A stable API is not subject to frequent changes, therefore leveraging the promised independence between API producers' and consumers' code. Changes in the API itself require changes in the API consumers' code because this code use services provided by the API. This situation might become problematic if changes to the API happen too often. Therefore, according to one software architect interviewed, APIs "*tend to be something well-thought out, and set in stone*," so that they are regarded as contracts with the clients (see section 4). As a result, API consumers expect that the API will not change often, and if it does happen, they also expect that these changes will not severely affect them. Recent work in software engineering tries to provide advice on how to properly change APIs so that the impact of those changes is minimized [11] [14].

3. RESEARCH SITE AND METHODS

Our fieldwork was conducted in a software development company named BSC (a pseudonym). BSC is one of the largest software development companies in the United States with products ranging from operating systems to software development tools, including e-business and tailored applications. The project studied, called MCW (another pseudonym), was responsible for developing a client-server application that had not yet been released during the period of the study. The project staff included 57 software engineers, user-interface designers, software architects, and managers, who were divided into five different teams, each one developing a different part of the application. The teams are designated as follows: lead, client, server, infrastructure, and test. The lead team was comprised of the project lead, development manager, user interface designers, and so on. The client team was developing the client side of the application, while the server team was developing the server aspects of the application. The infrastructure team was working in the shared components to be used by both the client and server teams.

Finally, the test team was responsible for the quality assurance of the product, testing the software produced by the other teams.

The MCW project (including its teams) is part of a larger company strategy focusing on software reuse. This strategy aims to create software components (each one developed by a different project) that can be used by other projects (teams) in the organization. Indeed, the MCW project uses several components provided by other projects, which means that members of the MCW teams need to interact with other software developers in other parts of the organization.

Regarding the data collection, we adopted non-participant observation [23] and semi-structured interviews [26], which involved the first author spending 11 weeks at the field site. Among other documents, we collected meeting invitations, product requests for software changes, and emails and instant messages exchanged among the software engineers. We were also granted access to shared discussion databases used by the software engineers. All this information was used in addition to field notes generated by the observations and interviews. We conducted 15 semi-structured interviews with members of all four sub-teams. The questions were designed to encourage the participants to talk about their everyday work, including work processes, problems, tools, collaboration and coordination efforts, and so on. Interviews lasted between 35 and 90 minutes. All the material collected has been analyzed using grounded theory techniques [33]. The grounded theory approach calls for an interplay between data gathering and analysis to develop an understanding of what is going on in the field and, most important, the reasons that explain what is going on. As the fieldwork progresses, hypotheses are generated and tested and modified according to the ongoing analysis of the data being collected. During our fieldwork, we eventually realized the fundamental role of APIs in the management of the interdependencies. Accordingly, we collected more information about this aspect in order to verify whether we had understood the software developers' work. Finally, the interviewees provided feedback on our interpretation of the roles of APIs in the process. This feedback was fundamental to improving our understanding of their work.

4. THE CONTRACTUAL NATURE OF APIs

4.1 Advantages

At the time of the study, BSC had recently adopted a strategy of developing reusable software components in which the concept of APIs had an important role. The underlying idea is that each software component would have a public and stable API through which its consumers could access the set of services provided by that component. APIs need to be public to allow other components to access the services its underlying component provides. They also need to be stable; that is, they cannot change very often. Otherwise, the expected reduced coupling between API consumers' and producers' code is not achieved. The importance of APIs in the coordination of the software developers was clearly recognized by members of the software development team, who agreed, *"APIs are the heart of the whole exercise."* As a member of the server team confirmed:

> "O*ur only work is to make these APIs work … the client team's* [work] *is to consume the APIs and create user interfaces.*"

Each software component and its respective API were developed by a different project team, and could be used by other projects teams in the organization. Most projects implemented different sets of services, therefore implementing different APIs. Furthermore, despite their willingness to reuse software components, different teams in the company developed different software components that provided similar sets of services because of reasons as implementation details, backward compatibility, different architectures, and so on. In this case, all these software components provide similar APIs. To guarantee that APIs were consistent and that software components were indeed reused, each project had a software architect responsible for the specification of the APIs. Weekly meetings of the organization's software architects were used to monitor this work.

Despite these meetings, the organization had no formally established process to create, implement, deploy, and maintain APIs. In one of the meetings that we observed, developers from different groups discussed the lack of recommendations by the organization's software architects on how to proceed when facing such issues. As one developer pointed out: "*All APIs need to look, feel, and smell the same.*" This lack of an established process had already been identified by the software architects and was starting to be discussed in the software architects' weekly meetings.

Although there was no formal process, an informal process was adopted by members of the MCW project. In this case, the majority of the APIs were developed by the server team, who provided services to be used by the client team. Each one of these APIs was specified by the server software architect as necessary. After an API was specified, it was discussed by all the interested parties in a formal design review meeting. The following people were invited to this meeting: the API consumers, the API producers, and the test team that eventually would test the software component functionality through this API. Another purpose of this meeting was to guarantee that the API met the requirements that the client team had and to make sure that API consumers understood how to use it and that the test team knew how to test it.

The API design review meeting exemplifies the first and foremost important role of any API: to establish a defined interface among, at least, two worlds. That is, APIs are *contracts* established between two parties. As such, they allow each party to go about doing its work without needing a huge deal of coordination among them. During the design review meetings, API producers, consumers, testers and other interested parties are all gathered together to reach a consensus about how the API is going to look. APIs are then negotiated. After this meeting, each party can work independently because they all expect that the established contract is going to be fulfilled. If, and when, this contract is broken, several problems arise (see next section).

API review meetings are also "meeting points" for all software developers interested in a particular API " where all developers would get to know each other. For instance, members of the test team meet the developers who will implement the API. Later, testers will email information to these API providers about how the APIs are going to be tested, with the intent of avoiding minor integration problems that could delay the development schedule. It is important to mention now that often the implementation of an API would not start right after these meetings. The meetings were planned in advance before anyone would need services from these

APIs. However, changes in people's roles and assignments during the software development process eliminated this knowledge about other software developers working in the same API. As we discuss in section 6 this lack of knowledge eventually brings coordination problems.

4.2 Disadvantages

The wide scope of the API design review meetings is problematic. As one software engineer who was interviewed informed us: "*The larger the audience, the wider the type of questions.*" In a similar fashion, members of the server team reported that client team developers want to understand too many implementation details of the APIs, instead of focusing on the "big picture" and using the meeting for clarification purposes only. The assumption of API providers is that consumers do not need to be aware of the implementation details of the API, everything else "behind" an API did not need to be disclosed.

As with any other contract, an important feature of an API is its stability, which means that APIs should not change often because when they change, the API consumers' code is affected as well. In other words, the impact of changing an API is high because it potentially leads to other changes in the source code. Despite that realization and all the discussion that takes place during an API design review meeting, APIs do change. This situation might be more or less problematic, depending on the type and amount of changes that occurr in the API. To minimize these problems, we noted that the server team (the API producers), before changing an API, met and negotiated these changes with the client team (the API consumers). We also noticed that, on some occasions, API consumers were notified about the changes, but these changes were not delivered to them right away. Because of that, API consumers were blocked because they needed this API to be able to work. Similarly, sometimes the changes to an API were not broadcast to the whole organization. Therefore, if there were other API consumers using this API, they would not be notified about them. A software engineer referred to this task of designing while an API was being changed as "*a total moving target.*"

The instability of some APIs was so evident that in some cases software engineers would ask questions such as: "*Is the* [name of a particular API] *changing*?' These questions were asked during the weekly meetings before these developers started working in the API, in order to avoid problems. It is important to notice that this instability is aggravated because current software development tools make it difficult to identify changes in APIs: no current tools support API diffing, versioning, updating, and so on. This lack of technological support makes these changes expensive and painful for API consumers. This issue will be discussed in section 8.

Note that software developers acknowledge that APIs need to change, therefore recognizing the inevitable situation where the API proposed in the design review is not the one being implemented. According to one of the developers:

> "*I've never seen a technical spec that describes functional requirements that has been implemented without changes.*"

> *"while you're developing code, everything can change.*"

Despite that, developers reported problems with changes in the APIs, and we observed several instances of complaints about this instability. Indeed, software developers at this organization faced a dilemma. They wanted to define APIs early in the process in order to allow independent work. However, at the same time, they wanted to avoid making the API unstable, which could be avoided only by postponing the definition of this same API.

5. SUPPORTING INDEPENDENT WORK

5.1 Advantages

Once an API is approved in the design review meeting, a first implementation is made available to its consumers through the configuration management tool. As mentioned before, APIs are interface specifications composed of sets of public classes, interfaces and methods, and the associated documentation (in this case, javadoc files). Besides these specifications, the software architect provides a shallow implementation of the API for the sole purpose of allowing the client team to immediately start programming against this API. According to one software architect:

> *"The first-pass delivery … is a shallow implementation, just enough to start some work."*

Software developers would refer to these dummy implementations as "local APIs," in contrast to "remote APIs," which are the real APIs implemented by the server team.[1] By adopting the usage of local APIs, it is possible to separate the work that each team needs to perform and temporarily remove dependencies among them. In the MCW project, the client team can start implementing against the local API while the server team can start implementing the (real) remote API.

Periodically, API providers replace parts of this shallow API implementation by their real implementation. For example, one participant stated:

> "*When it* [the implementation] *is ready, I replace the dummy code for the real implementation.*"

The parts to be replaced are often based on the suggestions provided by and the needs of the API consumers, according to the planned schedule. However, in order to do that, it is necessary to have knowledge about who are the API providers and consumers, which did not happen all the time in the organization.

5.2 Disadvantages

Once the parties involved agreed upon the APIs and "local" APIs are created, software development work proceeds independently. However, our data shows that problems arise due to those local APIs. Indeed, as reported by a software architect:

> *"This* [the usage of dummy implementations] *works to some extent. But as you push further along implementation dummy stuff starts not working. So, for example, the* [user-interface component] *list displays stuff, just dummy stuff, that works, but as soon as you want to open one of those dummy stuff, there is no stuff behind the dummy stuff so the list cannot hand off to the launcher* [another user-interface component] *that cannot hand off to the* [component] *you cannot open up*

[1] These APIs are called "remote" because when the application is released, they will be located in a remote machine. Note that "local" and "remote" APIs are the same; the unique distinction between them is their *implementation*.

because there is really nothing that far. It is a matter of how deep does the dummy stuff go. You really dive a bit, and then there is no more there. It kind of works in the start but as you go further along …"

To avoid this situation, software developers and managers perform an assessment of the local APIs in their weekly meetings. Sometimes it is possible for API consumers to continue using the local APIs for one more week. However, if the remote API is needed, the manager will contact the other team manager about it. Furthermore, the manager will suggest that the software engineer who is the API consumer contact his or her API provider. Note that the assumption here is that the API consumer knows who his or her API provider is, but this does not always hold (see section 6).

Replacing local APIs by remote APIs is theoretically a simple matter due to the usage of APIs. When this happens, the API consumer's code is being integrated with the API provider's code. However, integration problems between the client and the server teams had happened before in the former integration period. This led the client and server teams to adopt a pre-integration phase before the "official" integration period. Furthermore, in order to minimize this problem, the manager of the server team also decided to allocate a new hire into the testing team to test the code to be integrated to avoid possible problems. This means that the coordination effort required to integrate code developed by different software engineers is recognized by the managers. This additional effort is necessary even for members of the same team. Indeed, in another situation we observed, the client manager recognized that a developer would not be able to meet the schedule because he had to integrate his source code with two different software developers.

6. REIFYING ORGANIZATIONAL BOUNDARIES

6.1 Advantages

Each software component being developed by the organization might provide different services, which will consequently be made accessible through different APIs. This means that APIs are purposefully created to be the external boundaries of a component. Because each software component is implemented by a unique software development team, APIs also define the interfaces among these software development teams. APIs can then be viewed as boundaries of the teams: they define the limits of what will be delivered and what needs to be done by each team. Being an API provider means being a member of the team that is implementing this API, and consequently understanding its implementation details. Conversely, being an API consumer means being part of a different team, which does not need to know the API implementation details[2]. APIs are then reifications of the already established team divisions. In other words, APIs reify organizational boundaries: any two (or more) given teams in the organization that need to interact (i.e., that their code needs to interact) will do so through the appropriate set of APIs that will link the software components they are developing. As Mintzberg discusses these organizational boundaries "work to the advantage of the organization, allowing each unit to give particular attention to its own special problems" [27]. Typically, complex components need to interact with several other components, meaning that several APIs will mediate the cooperation among these components and, consequently, among members of these teams. For instance, the architects that we interviewed reported that there are at least six different APIs mediating the work between the MCW client and server teams.

To summarize, APIs achieve an organizational goal, which is to separate teams of developers so they can be named, organized, managed, and so on. This is especially useful, and indeed planned, during the initial phases of software construction because they allow software engineers to work independently without worrying about the impact of their colleagues' work in their own work. However, it might create problems in the later stages of this process, as discussed in the next section.

6.2 Disadvantages

In contrast, because APIs are reifications of organizational structures, they divide the work necessary to develop software into two distinct parts: an internal part responsible for implementing the API, and an external part responsible for using this same API. As a side effect of the isolation provided by APIs, we noticed that teams lacked awareness about other teams' work. In the MCW team, this problem was remedied by the managers, who maintained constant and intensive communication about their teams' progress and schedules. Additionally, another approach adopted by these teams was to pair developers (one from each team) according to the APIs. That is, for each server team member responsible for implementing an API, there was a client team member who was the consumer for that API. This organizational solution failed because API consumers did not want to appear to be pressuring their server developer counterparts. Similarly, we found out that in the server team, some software engineers were not aware of their client counterparts, i.e., those who would consume the API they were implementing. According to the software architect interviewed:

"In our team meeting yesterday and other ones… people seem to be reluctant to talk to their counterparts too much … in the sense that they feel they're bugging the other person … and that is a problem because, I mean, the reason why we are here … the reason we're getting paid, we are developing a product and that interaction needs to happen."

One might think that this type of knowledge about their counterparts is not necessary during the initial stages of development while team members might still work independently. However, as a software architect pointed out, this is still problematic:

"People thinking there's somebody else doing something [on the API] *and when, you know* [the API is needed] *… it is an empty void because they did not step up and said: 'I tried to identify my server counterpart or my client counterpart or if there is anyone. We got a problem here!'"*

[2] It is possible to use APIs to coordinate the work of software developers in the same team. However, this is the exceptional case. Indeed, we did not find any instance of this situation in the MCW project.

The "isolation" created by APIs also hindered the collaboration among members of different teams that were not paired. For instance, another team in the organization was responsible for implementing a component that provided services for both the server and the infrastructure team. Due to the isolation of the teams, members of these teams were not aware of this common dependency, therefore they were working in parallel in overlapping aspects. One software engineer identified this issue and decided to talk to the members of the other team so that they all could align their efforts and avoid duplicate work.

7. DISCUSSION

In our work, we identified two major roles played by APIs in the software development process in the context of the MCW and its hosting organization: they function as contracts among stakeholders and, in addition, as reifications of the organizational boundaries. Our overall findings show that APIs simultaneously allow and constrain collaboration among software developers, contradicting the common wisdom among software developers that APIS are only beneficial.

On the one hand, APIs facilitate collaboration during the process of breaking a system into pieces that can be developed independently (according to the information-hiding principle [30]) because (i) as reifications, they enforce the organizational boundaries of team membership, and (ii) as contracts they establish a shared understanding of what needs to be done and at some level formalize this agreement. By breaking the system into pieces, it is possible to isolate development work, allowing software engineers to work without being affected by their colleagues' work. This need for isolation is a common theme in software engineering because of the several interdependencies that occur in these efforts. To mention only one example, Sarma and colleagues [31] discuss how configuration management tools support this isolation. A problematic situation arises when APIs are not seen as contracts, which allows them to be changed, causing disruptions to the collaborative process (see section 4.2).

On the other hand, APIs limit collaboration during the process of recomposition—the work of putting all pieces together to create a software artifact [16]. This happens because the initial process of decomposition creates social relationships among the stakeholders that need to be maintained during the whole software development process; otherwise the software cannot be later recomposed [*Ibid.*]. This was already recognized by Parnas who defined a software module as "a responsibility assignment rather than a subprogram" [30].As discussed in section 5.2, despite the fact that APIs were used as contracts, problems arose during the code integration developed by API consumers and providers. Similarly, section 6.2 shows that APIs (seen as reifications) hinder collaboration among software developers from different teams, eliminating opportunities for cooperation.

At this point, it is important to mention that over 30 years ago Conway [6] had already recognized that the structure of the system mirrors the structure of the organization that designed it, a relation known as Conway's Law. Our findings that APIs reinforce the organizational boundaries confirm this idea. However, we go beyond that by explaining why these boundaries need to be somehow flexible to allow inter-team collaboration.

One way of providing this flexibility is through the concept of awareness as proposed by Dourish and Bellotti [12]. Several studies have discussed the role of awareness of others' actions in facilitating coordination of individuals in settings as varied as ship bridges [21], aircraft cockpits [22], and transportation control rooms [19]. In particular, recent work has shown the importance of awareness in software development as well (see [9], [18], [16], and [34]). Based on this empirical evidence, tools have been built in the last few years to support this approach (such as Jazz [3, 4], Palantír [31] and Night Watch [29]). This study builds upon this previous work by providing information about *what* information about others' actions software developers need to be aware of. That is, API consumers need to be aware of changes in the API that they are using because the code that they are writing depends on it. Indeed, people are not interested in all information that is provided to them. As Schmidt [32] points out:

> *"(...) in depending on the activities of others, we are 'not interested' in the enormous contingencies and infinitely faceted practices of colleagues unless they may impact our own work ... An actor will thus routinely expect not to be exposed to the myriad detailed activities by means of which his or her colleagues deal with the contingencies they are facing in their effort to ensure that their individual contributions are seamlessly articulated with the other contributions."*

Because an API is usually written by a developer from a different software development team, this field study also suggests that software developers need to be aware of actions from his colleagues from other teams. In this case, this is necessary because a dependency exists between API consumers and providers, which is the API itself. This finding is in contrast to what Grinter [16] suggests in her discussion about organizational awareness: team information should be aggregated. Our results suggest that individual information about team members' actions is also necessary. Therefore, this field work also illuminates the list of people that one needs to be aware of, which includes not only their teammates but other software developers in the organization as well. We can certainly imagine that software developers need to be aware of other colleagues that they depend on as well.

Furthermore, our data show that APIs are successful in facilitating collaboration because they hide details that software engineers do not need to know at a particular moment in the process. This suggests that awareness needs to be balanced with the need for isolated and independent work. In other words, private and public work are both necessary in cooperative software development [9].

Finally, it is important to mention that Grinter [17] discusses how software architects need to convince other members of the organization to "buy into" their design. We noticed this same phenomenon in our field study: software architects bring into the API design review the client team and other potential API consumers, so that they can approve the API design. That is, the API design is a technical process as much it is a social process involving communication, coordination, and negotiation [2]. One can not reconstruct an API based solely on technical decisions, there are also social conditions that somehow are embedded in the API itself.

8. IMPLICATIONS FOR TOOLS

In the previous section, we discussed how APIs play a dual role in the coordination of collaborative software development. Moreover, we discussed how APIs are reification of organizational boundaries in the organization, therefore allowing and constraining collaboration among software developers of different teams. One of the reasons why they hinder collaboration is because they do not allow software developers' to be aware of their colleagues' actions that affect their work. Because not all actions are important, we argue that awareness tools need to be able to hide some details, while at the same time, provide useful information to let software developers align their work. This suggests that translucent approaches could be very useful in the design of collaborative software development tools because their goal is to make social information available without, however, presenting too much information [13]. Furthermore, our field study also suggests how to go about deciding which information should be presented and which information should be hidden. Information about changes that might affect another's person work should be presented to the interested parties, while everything else can be either hidden or presented in a less intrusive way. Information about changes that do not affect directly one's work is still interesting. For example, de Souza, Redmiles and Dourish [9] present empirical evidence that this information is useful to software developers learn about each other's areas of expertise, therefore leveraging expertise identification when necessary.

Finally, we argue that the source-code itself can be an important resource to identify social dependencies since it contains information about the technical dependencies among pieces of software and these suggest the social relationships that need to be built among software developers to facilitate the integration process. For instance, dependencies among pieces of code exist because components make use of services provided by other components: let's say that a component *A* uses the services of another component *B*, as a result, *A* depends on *B*. Assuming that *A* is being developed by engineer *a* and *B* is being implemented by engineer *b*, we similarly find that engineer *a* depends on *b*. A data structure containing all the dependency relationships of a software application is called a call-graph, because it contains information of which components *call which* other components. Information from this call-graph can be used to describe the technical dependencies in one software application. This needs to be combined with authorship information about each component or module to allow the identification of social dependencies. Configuration management repositories contain this authorship information since they track the changes made to each component alongside the information about the developer who performed the change [5]. Combining information from the call-graph with authorship information can then create a "social call-graph", which describes which software developers depend on which other software developers for a given piece of code. Figure 1 below presents an example of a "social call-graph" from a software development project being conducted at UCI called Ariadne. A directed edge from package A to B indicates a dependency from A to B. Directed edges between authors and packages indicated authorship information. Note that authors are leaves in this graph.

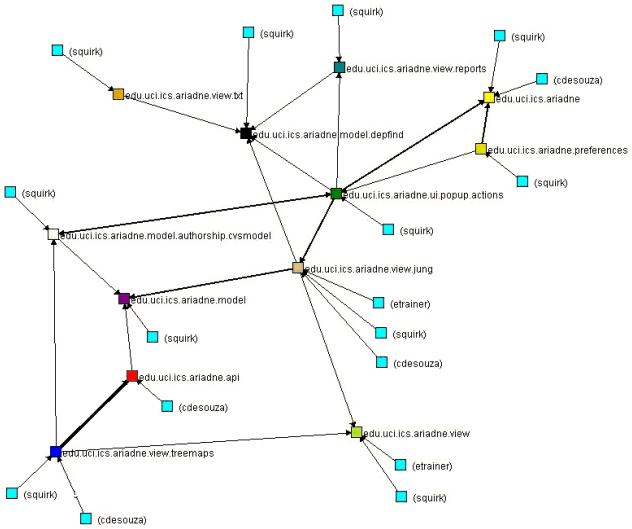

Figure 1: An example of a "social call graph"

We argue that current software development tools have only focused on the call-graph itself. For example, this information can be used to calculate different metrics that provide useful information about the organization of the source-code. For example, one can evaluate the cohesion and coupling of the modules using information from the call-graph. Unfortunately, information from "social call-graphs" have not been explored yet. We believe that such graph is a potential resource that could be used for a variety of purposes. For example, this "social call-graph" could be used to provide more selective information about software developers' actions. In our previous example, developer *a* needs to be aware of *b*'s actions regarding changes in the interface of the component B. That is, if *b* changes the implementation of the services that *B* provides, it is necessary to inform developer *a* of these changes. Furthermore, this "social call-graph" could be used by software developers to identify other developers with similar interests, as in the situation described in section 6.2 where developers who shared a dependency where performing duplicate work because they were not aware of each other. This approach is similar to the one adopted in the ExpertiseBrowser system [28], that provides expertise identification. In both approaches, it is necessary to associate software engineers with the pieces of software that they produce. However our approach also requires the association of pieces of software among themselves according to their dependencies. This idea is based on the actor-network approach proposed by Latour [25], where networks of artifacts and human (both called "actants") are represented together.

Because of the information that they have available, "social call-graphs" could easily generate social network graphs describing the dependency relationship among software developers, in this case *without* depicting dependencies among software components. Figure 2 below presents an example of a "social call-graph." This example is based on information collected from the MCW team through our interviews and non-participant observation. Members of the client team are represented by *cN*, where N is an integer from 1 to 8. Similarly, members of the server team are represented by *sN*, and finally, members of test team are presented by *tN*. The other letters (*n, d* and *a*) indicate other teams in the organization. Arrows indicate dependency relationships from the source to the

target of the arrow, for example, developer *c2* depends on developer *s1*.

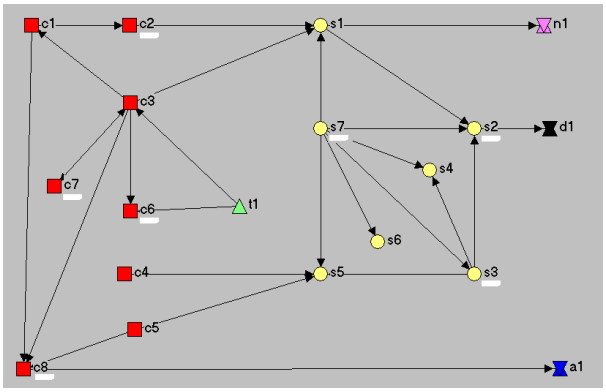

Figure 2: An example of a social network graph describing dependency relationships among software developers

The next step then is to analyze these graphs with social networks algorithms in order to assess potential coordination problems in the software development process. For example, one could generate technical recommendations about how to reorganize the source code, or provide managerial recommendations about how to change the division of labor to minimize the coordination effort of some developers that have to deal with too many dependencies. The construction and analysis of "social call-graphs" and social network graphs generated from those are our next steps and are briefly described in the next section.

9. CONCLUSIONS AND FUTURE WORK

This paper described a field study that examined the use of application programming interfaces (APIs) in the management of interdependencies in cooperative software development. The notion of APIs is a fairly familiar concept in software engineering and among professional software developers. They are technical constructs that instantiate the principle of information hiding [30], aiming to create well-defined interfaces between two pieces of software to minimize the dependency between them. Usually, these two pieces of software are developed by two different teams of software developers, and as a result APIs also reduce the dependency between the work of these teams. Our results suggest that APIs facilitate the coordination of activities in software development because they can be seen as (i) contracts among stakeholders, and (ii) as reifications of organizational boundaries. That is, APIs also achieve the organizational goal of maintaining teams of software developers isolated from each other. However, by doing that, they reduce the opportunities for cooperation, consequently disrupting the collaborative construction of software, especially during the integration period. To be more specific, the main problem created is the lack of awareness about other colleagues' actions. In order to minimize this problem, we also described how collaborative software development tools could be extended to make use of the artifact being built (the software itself) to facilitate collaboration. This is based on the observation that (i) dividing a software system is simultaneously a division of labor [30], and (ii) a software system becomes embedded with part of the social relationships that surround its construction [6, 16]. Currently, we are working on the development of Ariadne, a collaborative software development tool based on the Eclipse IDE, that adopts this approach. We plan to deploy this tool among professional software developers to find out whether it is useful in coordinating their work.

ACKNOWLEDGMENTS

The authors thank CAPES (grant BEX 1312/99-5) and NASA/Ames for financial support. This effort was sponsored by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-00-2-0599. Funding was also provided by the National Science Foundation under grant numbers CCR-0205724, 9624846, IIS-0133749 and IIS-0205724. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), the Air Force Research Laboratory, or the U.S. Government.

10. REFERENCES

[1] "Application Programming Interfaces," vol. 2004: Software Engineering Institute - Carnegie Mellon University, 2003.

[2] Bucciarelli, L. L., *Designing Engineers*. Cambridge, Ma: MIT Press, 1996.

[3] Cheng, L.-T., De Souza, C. R. B., et al., "Building Collaboration into IDEs. Edit -> Compile -> Run -> Debug ->Collaborate?" in *ACM Queue*, vol. 1, 2003, pp. 40-50.

[4] Cheng, L.-T., Hupfer, S., et al., "Jazz: a collaborative application development environment," ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications, pp. 102-103, Anaheim, CA, USA, 2003.

[5] Conradi, R. and Westfechtel, B., "Version Models for Software Configuration Management," *ACM Computing Surveys*, vol. 30, pp. 232-282, 1998.

[6] Conway, M. E., "How Do Committees invent?," *Datamation*, vol. 14, pp. 28-31, 1968.

[7] Curtis, B., Krasner, H., et al., "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, pp. 1268-1287, 1988.

[8] de Souza, C. R. B., Redmiles, D., et al., "Management of Interdependencies in Collaborative Software Development: A Field Study," International Symposium on Empirical Software Engineering (ISESE'2003), pp. 294-303, Rome, Italy, 2003.

[9] de Souza, C. R. B., Redmiles, D. F., et al., ""Breaking the Code", Moving between Private and Public Work in Collaborative Software Development," International Conference on Supporting Group Work (GROUP'2003), pp. 105-114, Sanibel Island, Florida, USA, 2003.

[10] des Rivieres, J., "Eclipse APIs: Lines in the Sand," in *EclipseCon*, vol. 2004, 2004.

[11] des Rivieres, J., "How to Use the Eclipse API," vol. 2004.

[12] Dourish, P. and Bellotti, V., "Awareness and Coordination in Shared Workspaces," Conference on Computer-Supported

Cooperative Work (CSCW '92), pp. 107-14, Toronto, Ontario, Canada, 1992.

[13] Erickson, T. and Kellogg, W. A., "Social Translucence: An Approach to Designing Systems that Support Social Processes," *Transactions on HCI*, vol. 7, pp. 59-83, 2000.

[14] Fowler, M., "Public versus Published Interfaces," *IEEE Software*, vol. 19, pp. 18-19, 2002.

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

[16] Grinter, R. E., "Recomposition: Putting It All Back Together Again," Conference on Computer Supported Cooperative Work (CSCW'98), pp. 393-402, Seattle, WA, USA, 1998.

[17] Grinter, R. E., "System Architecture: Product Designing and Social Engineering," Work Activities Coordination and Collaboration, pp. 11-18, San Francisco, CA, USA, 1999.

[18] Grinter, R. E., "Using a Configuration Management Tool to Coordinate Software Development," Conference on Organizational Computing Systems, pp. 168-177, Milpitas, CA, 1995.

[19] Heath, C. and Luff, P., "Collaboration and Control: Crisis Management and Multimedia Technology in London Underground Control Rooms," *Computer Supported Cooperative Work*, vol. 1, pp. 69-94, 1992.

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

[21] Hutchins, E., *Cognition in the Wild*. Cambridge, MA: The MIT Press, 1995.

[22] Hutchins, E., "How a Cockpit Remembers its Speeds," *Cognitive Science*, vol. 19, pp. 265-288, 1995.

[23] Jorgensen, D. L., *Participant Observation: A Methodology for Human Studies*. Thousand Oaks: SAGE publications, 1989.

[24] Larman, G., "Protected Variation: The Importance of Being Closed," *IEEE Software*, vol. 18, pp. 89-91, 2001.

[25] Latour, B., "Where are the missing masses? The sociology of a few mundane artifacts.," in *Shaping Technology / Building Society: Studies in Sociotechnical Change*, W. Bijker and J. Law, Eds. Cambridge, MA: MIT Press, 1994, pp. 225-258.

[26] McCracken, G., *The Long Interview*: SAGE Publications, 1988.

[27] Mintzberg, H., *The Structuring of Organizations: A synthesis of the research*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[28] Mockus, A. and Herbsleb, J. D., "Expertise Browser: A Quantitative Approach to Identifying Expertise," International Conference on Software Engineering, pp. 503-512, Orlando, FL, USA, 2002.

[29] O'Reilly, C., Morrow, P., et al., "Improving Conflict Detection in Optimistic Concurrency Control Models," 11th International Workshop on Software Configuration Management (SCM-11), Portland, Oregon, 2003 (to appear).

[30] Parnas, D. L., "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM*, vol. 15, pp. 1053-1058, 1972.

[31] Sarma, A., Noroozi, Z., et al., "Palantír: Raising Awareness among Configuration Management Workspaces," Twenty-fifth International Conference on Software Engineering, pp. 444-453, Portland, Oregon, 2003.

[32] Schmidt, K., "The critical role of workplace studies in CSCW," in *Workplace Studies : Recovering Work Practice and Informing System Design*, P. Luff, J. Hindmarsh, and C. Heath, Eds.: Cambridge University Press, 2000, pp. 141-149.

[33] Strauss, A. and Corbin, J., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*, Second. ed. Thousand Oaks: SAGE publications, 1998.

[34] Teasley, S., Covi, L., et al., "How Does Radical Collocation Help a Team Succeed?," Conference on Computer Supported Cooperative Work, pp. 339-346, Philadelphia, PA, USA, 2000.