How a Good Software Practice Thwarts Collaboration – The multiple roles of APIs in Software Development

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

The principle of information hiding has been very influential in software engineering since its inception in 1972. This principle prescribes that software modules hide implementation details from other modules in order to decrease their interdependencies. 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 qualitative study on how practitioners use APIs in their daily work. Although particularly interested in aspects of collaboration, we report all findings about their observed use. The findings include mundane observations that are predicted by theory, ways that APIs support collaborative software development. But the findings also include some surprises, ways that APIs hinder collaboration. The surprises indicate directions for further improvement of collaborative software development practices and 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

Empirical software engineering, qualitative studies, interfaces, application programming interfaces.

1. INTRODUCTION

It has been long recognized that breakdowns in communication and coordination efforts constitute a major problem in collaborative software development [7]. One of the reasons is the large number of interdependencies among activities in the software development process, among different software artifacts, and finally, within different parts of the same artifact. To overcome this problem, the field of software engineering has developed tools, approaches, and principles to deal with interdependencies. Configuration management and issue-tracking systems are examples of such tools, while the adoption of software development processes ([2, 31], and [15]) exemplifies an organizational approach [8, 9]. One of the most important and influential examples principles used to manage dependencies is the idea of information hiding proposed by Parnas [33]. According to this principle, software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" [27]. Information hiding aims to decrease the dependency (or coupling) between two modules so that changes to one do not impact the other. This principle is instantiated as several different mechanisms in programming languages that provide flexibility and protection from changes, including, data encapsulation, interfaces, and polymorphism [27]. In particular, separating interface specifications from their implementation is a growing trend in software design [14]. Furthermore, interface specifications are believed to be helpful in the coordination of developers working with different components [17]:

> *"(…) 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-specified commitments about how the two will hook together."* [emphasis added]

APIs (application programming interfaces) are a common way of hiding component specification and implementation details from users of those components (e.g. see [10]). They are commonly used in the industry to divide software development work, including *distributed* software development, and are widely regarded as "the only scalable way to build systems from semi-independent components" [17].

This paper describes an empirical study that we performed on how practitioners use APIs in their daily work. Although particularly interested in aspects of collaboration, we report all findings about their observed use. The findings range from mundane observations that are predicted by theory to surprising observations. As predicted by theory, APIs serve as *contracts among stakeholders, reifications of organizational boundaries,* and as *a common language among software developers.* In these ways, APIs support developers collaborating on a common project yet allow them to work independently in isolation. However, there is a surprising side effect we observed, that the isolation hinders some 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 reviews concepts surrounding APIs and explains their adoption by industry. After that, section 3 presents the research site studied and methods that we used in our study. Then, Section 4 describes how the organization and teams we observed go about developing, using, and maintaining APIs. Section 5, describes the multiple roles played by APIs in a collaborative software development process. Section 7 and 8 respectively discuss the data collected and the implications of our findings to the design of CSCW tools. Finally, conclusions and ideas for future work are presented.

2. APPLICATION PROGRAMMING INTERFACES

In order to understand the concept of application programming interfaces, we need to understand a couple of important software engineering principles first. Separation of concerns, for example, is one the most important principles in software engineering that allow one to deal with different individual aspects of a problem, so that it is possible to concentrate on each separately. When different parts of the same system are dealt separately, we are talking about a type of separation of concerns named modularity [16]. Modules should be designed according to the information hiding principle proposed by Parnas [33]. According to this principle, software modules should be both "open (for extension and adaptation) and closed (to avoid modifications that affect clients)" [27]. This principle is instantiated in programming languages as several different mechanisms such as data encapsulation, interface specifications, and polymorphism [27]. An application programming interface (API) is defined by the Software Engineering Institute as:

> *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* [1].

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. Therefore, we will adopt a more loose definition of an API as proposed by des Rivieres [10]: an API is a well-defined interface that allows one software component to access programmatically another component and is normally supported by the constructs of programming languages. 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). 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 or components. These two (or more) applications are often developed by different teams, and hardly ever individuals. An example of a well-known API is the Microsoft Windows API that allows a program to access and use resources of the underlying operating system such as file system, scheduling of processes, and so on.

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. That is, modularity, and consequently separation of concerns, is achieved.

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 the API producers' and consumers' code. Changes in the API itself require changes in the API consumers' code because this code uses 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 5.1). 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 that we will call 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), is responsible for developing a client-server application that had not yet been released during the period of the study. The project staff includes 57 software engineers, user-interface designers, software architects, and managers, who are 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 side of it. 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 [24] and semi-structured interviews [28], 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, collaboration and coordination efforts, problems, tools, and so on. Interviews lasted between 35 and 90 minutes. All the material collected has been analyzed using grounded theory techniques [36]. 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 DEVELOPMENT OF APIs

4.1 The Organizational approach to APIs

At the time of the study, BSC had recently adopted a strategy of developing reusable software components. 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 another member of the server team confirmed:

> *"Our 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 several APIs. Despite their willingness to reuse software components, different teams in the company developed different software components that provided similar sets of services. For example, one team would provide access to email services implemented in one particular platform. Another team would also provide access to email services in a different platform. In this case, these software components would provide similar APIs. To guarantee that APIs were consistent and that software components were indeed reused throughout the organization, each project team 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.

4.2 The Development of APIs

Despite these meetings, the organization had no established formal 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 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 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.

4.3 "Local" and "Remote" APIs

Once APIs are reviewed, they are made available through the configuration management tool to their consumers. As mentioned before, APIs are composed of sets of public classes, interfaces and methods, and the associated documentation (in this case, javadoc files). Besides that, the software architect defining the API 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 but it does not really flesh out anything."*

Software developers would refer to these implementations as *local* APIs in contrast to *remote* APIs, which are the APIs implemented by the server team. These APIs are called remote because when the application is released, they will be located in a remote machine. That is, the local and the remote APIs are the same; the distinction between them is solely based on the functionality provided by the current implementation.

Periodically, API providers replace parts of this shallow API implementation by its real implementation often based on suggestions provided by and needs of the API consumers. But also, according to the planned schedule to the API:

> "(…) *when it* [the implementation] *is ready, I replace the dummy code for the real implementation*"

By adopting this approach, the organization could separate the work that each team needs to perform and temporarily remove dependencies between the teams: the client team can start implementing against the local API, while the server team can start implementing the (real) remote API. Work can now proceed in parallel. Hopefully, replacing local APIs by remote APIs is a simple matter. However, our data shows that is a problematic aspect. These problems are discussed in section 6. The following section describes the multiple roles played by the API in this particular team and organization.

5. THE MULTIPLE ROLES OF APIs

The previous sections described how APIs are used by the MCW project team and other parts of the organization in order to manage their interdependencies and successfully cooperate. Using grounded theory techniques we analyzed our field notes and the transcribed interviews to find out the roles played by APIs in the software development process. We found that they play three different roles; each one of them is described below.

5.1 APIs as Contracts

APIs review meetings exemplify 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 while minimizing the coordination needs between 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 like. After this meeting, each party can work independently because they all expect that the established contract is going to be fulfilled. These meetings are also "coordination events" for all software developers interested in a particular API. 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. On the other hand, if the scope of these meetings is too wide, they might be problematic. One software engineer interviewed informed us that: "*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 in the "big picture" and using the meeting for clarifications purposes only.

Seeing APIs as contracts also allows one to notice how APIs specify the set of services a component is going to provide. Because they are the only updated document containing the description of the services to be provided by a component, they often play the important role of specification documents. 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.*"

Because of time pressure and other constraints, official specification documents are often outdated. Indeed, as one software developer noticed, some teams would even write their specs as APIs calls. In several occasions during our fieldwork, software engineers were observed inspecting the APIs to figure out the services offered by a component. Later, this observation was confirmed during the interviews.

The role of APIs as specification documents was so evident that one developer during an interview reported that APIs are equivalent to functional requirements: they describe small pieces of work, function calls and how to aggregate them to do something useful. He even defined APIs in regard to requirement specification documents: "[a] *well defined set of interfaces to allow the requirements to be met*".

5.2 APIs as Organizational Boundaries

Each software component being developed by the organization might provide different services, which will 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 necessary interfaces between these software development teams. APIs can be thought as boundaries of the teams: they define the limits of what can be known about and what needs to be done by each team. Being an API provider means to be a member of the team who is implementing this API, and consequently to understand its implementation details. On the other hand, to be an API consumer means to be part of a different team, which does not need to know the API implementation details[1]. APIs are then reifications of the already established team structures. That is, APIs reify organizational boundaries: any two given teams that need to interact (i.e., that their code needs to interact with each other) in the organization will do so through the appropriate set of APIs that will integrate their software components.

Typically, complex components need to interact with several other components, meaning that several APIs will mediate the cooperation among members of these two 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. In addition, each one of these teams needed to interact with other teams in the organization because their components needed to somehow interact.

[1] Sometimes APIs are used to coordinate the work of software developers in the same team. That is, dependencies *within* a software development team might be handled through the usage of APIs. However this is the exceptional case.

5.3 APIs as Communication mechanisms

Finally, APIs allowed the software developers to talk about their work while performing it. More specifically, it allowed them to talk about their dependencies. For example, the following quote resumes the division of labor between the client and server teams:

> *"Our only work is to make these APIs work … the client team's* [work] *is to consume the APIs and create user interfaces"* [member of the server team]

This is a typical thing among software developers, who find it useful to associate components with the teams developing them [18]. In this case, teams are associated with the APIs that they are developing.

6. ON THE LIMITATIONS OF APIs

This section describes some problems that the MCW project team faced during this field study. We describe these problems according to the role played by the APIs. The first three problems were associated with the contractual nature of APIs, while the fourth one is associated with the fact that APIs reinforce organizational boundaries. Each one of these problems is discussed below.

6.1 Instability

An important feature of any contract, and consequently any API, is its stability, which means that APIs should not change often because when they change, their consumers need to make changes in their code as well. In other words, changing an API has a high impact because it potentially leads to several other changes in the source code. Despite that and all the discussion that takes place during the API design (e.g., during the API review meetings), APIs do change. As one would expect, these changes impact API consumers. For instance, the client team needs to update its code. This situation might be more or less problematic depending on the type and amount of changes that occurred in the API. To minimize these problems, we observed that members of the server team (the API producers), before changing an API, meet and negotiate these changes with members of the client team (the API consumers). Furthermore, we noticed that both teams do not adopt any technical support to this: if there are tools available to perform API versioning and diff, they are not used by these teams. Finally, our last observation is that in some occasions the client developers would be notified about changes in the API, but the actual changes to the API were not delivered to them right away. As explained before, the client team needs the server APIs to be able to use them as "local" APIs, creating a temporary independence from the server's team. In some cases, the changes to the API are not spread in the organization. And since other teams also depend on the set of services that the component through its API makes available this situation makes the design and implementation tasks much harder. As one software developer reported: "*this* [the task of designing using an evolving API] *is a total moving target*".

The instability of some APIs was so evident that developers often would ask questions like: "*Is the* [name] *API changing?*'. They would ask this question in their weekly meetings before starting to work in the API in order to avoid problems. This instability is aggravated because current tools make it difficult to identify changes in APIs.

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

> "*I've never seen a technical spec that describes functional requirements that have been implemented without changes.*"
>
> "*while you're developing code, everything can change.*"

Despite that, several developers reported problems with changes in the APIs.

6.2 Incompleteness

APIs are widely used in this company to facilitate the coordination of teams of developers because of the agreed upon common and well-defined interface that can be used to connect these teams' source-code. The expectation then is that the integration should go smoothly. The following quote from one of the managers clearly reflects this expectation: "*if we use* [N] *weeks for integration, then we're doing something wrong*".

However, in reality, problems arise during the integration period. In the MCW team, for example, several problems happened during the last organizationally scheduled integration period. This situation led both the client and server teams to adopt a "pre-integration" period before the official integration period. Furthermore, the manager of the server team decided to assign a new hire to perform "smoke tests" to minimize possible integration problems.

As mentioned before, the adoption of "local" and "remote" APIs is another approach adopted by the organization to facilitate the integration. However this approach is also limited, it only works in the early stages of development becoming problematic as work progresses:

> *"I think … 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 list displays stuff, just dummy stuff, that works, but as soon as you want to open one of those dummy stuff, there is no stuffy behind the dummy stuff so the list can not hand off to the launcher* [component] *that can not hand off to the* [component] *…you can not open up because there is really nothing that far…It is a matter of how deep does the dummy stuff goes. You dive a really bit and then, there is no more there. It kind of works in the start but as you go further along (…)"*

In order to avoid this situation, assessment of the local APIs is performed by managers, software architects and software developers during their weekly meetings. The goal is basically to assess either if the "local" API can still be used (work can proceed independently) or if it is time to use the "remote" API (work now has to be integrated). Sometimes, it is possible to API consumers to continue using the "local" APIs, which means that they will go on working without contacting their API providers. However, when the "remote" API is needed, the manager will contact the other team manager and suggest the API consumer to contact his or her API provider. Note that there is an assumption here, which is that the API consumer knows who his or her API provider is, which does not always hold. Indeed this problem is described in the next section.

The main reason why there are problems during the integration period is that, despite the amount of effort spent in the design of an API, it is necessarily incomplete. That is, an API defines the syntactic aspects of an interface, however, it does not provide enough details about its implementation, and sometimes these details are necessary to the API consumer [25, 26] (see the discussion section). The important point then is how this incompleteness increases the already existing coordination problems in any collaborative software development process.

6.3 Lack of Awareness

APIs, and most generally interfaces, 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. That is APIs reify the organizational structures that define the team boundaries. Again, this brings several advantages to the coordination process (see section 5.2). However, 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, an approach adopted by the MCW client and server 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 in some occasions 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 software development while team members might still be able to work independently. However, as a software architect pointed out, this lack of awareness is still problematic:

> *"People think 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!'"*

Note that the API design review meetings play an important role in the coordination work because that is when all software developers interested in a particular API meet potentially avoiding some of the problems described above. However, the design of the API, and consequently the API design review meetings, occur well-before anyone needs the services provided by the API. That is, often the implementation of an API will not start right after its design review. Changes in people's roles and assignments during the software development process therefore remove this knowledge about API consumers and producers.

In addition, by reifying organizational boundaries APIs indirectly 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. Members of these teams were not aware that they shared this dependency and were working in parallel in overlapping aspects of this task. 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 general, the advantages and disadvantages of APIs for supporting collaborative software development are related to the independent work (or isolation) they support. This need for isolation is a common theme and a recurrent problem in software engineering. For example, Grinter, Herbsleb and Perry [17] describe how software development projects coordinate their efforts across multiple locations. One of the models described by these authors is called product structure and uses standards and interface specifications as coordination mechanism. One of the problems faced by the projects adopting this model was that components evolved independently making it hard to align features during the integration period. In our field study, while we identified the same coordination role played by interface specifications (APIs), we found out that the independence created by these interfaces was seen in the organization as an advantage, not as a problem. On a different approach, Sarma and colleagues [35] discuss how configuration management tools support both good and bad isolation. Good isolation occurs is when these tools allow software developers to work independently without being affected by their colleagues' work. Bad isolation, on the other hand, happens when these tools decrease the awareness software developers have of their colleagues' relevant work, potentially leading to coordination problems. These authors draw on the concept of *awareness* [12] to augment configuration management tools with visualization of other developers' activities [35], therefore supporting good isolation and minimizing the problems of bad isolation. Awareness of others' actions has been recognized as an important aspect that facilitates coordination of individuals in settings as varied as ship bridges [22], aircraft cockpits [23], and transportation control rooms [21]. Recent work, including this present work, has shown the importance of awareness in software development efforts (see [9, 18, 20], and [37]). Some trials at supporting awareness in software tools has begun as well (e.g., Jazz [5, 6], Palantír [35] and Night Watch [32]). Closely tied to the work on awareness, Ericksson and Kellogg define a further refined concept, *social translucence* [13]. Their aim is to facilitate the design of systems that support communication and collaboration among large group of people. Translucent systems make people's actions visible through intelligent user interfaces and thereby facilitate awareness among collaborators.

One could argue that process centered software environments ([2, 31], and [15]) support awareness and social translucency because these environments normally represent where documents (or software components) need to be delivered to next. However, these systems are not focused on revealing to one participant which other participants might be involved in a process. Indeed, the point of employing some of these systems is to generalize or hide such information which is seemingly not pertinent to an individual's particular activity. In short, the level of detail needed with respect to the kinds of awareness and translucency discussed here may not be appropriate to process tools.

Notions such as translucency and awareness indicate an alternative to simply delaying the definition of APIs. The underlying idea is that *private work* has *public consequences,* and vice-versa [9]. In this study, this means that the work performed by API providers *impacts* the API consumers' work (if the API changes for example). And, at the same time, the way API consumers' use the API influence how the API should be designed by the API providers.

Interestingly, a distant analogy to social translucency was once discussed in the programming language and software engineering arenas in the idea of *open implementation* [25, 26]. Namely, Kizcales and colleagues recognized the need to make APIs less opaque in a sense. They maintained that users of software components needed to know some information about a component's implementation and not just its API in order to make appropriate decisions about whether to use the component and how to use it. In these terms, our goal of supporting improved collaboration would mean extending the concept of openness to reveal the network of people relevant to making decisions about the API. This topic is discussed in more details in the next section.

In their seminal field study of software development projects, Curtis and colleagues observed the following problems: "the thin spread of application domain knowledge, fluctuating and conflicting requirements, and communication bottlenecks and breakdowns [7]." Our findings are primarily related to the last two issues because we carried out the study with a focus on collaboration and communication issues. In these aspects our findings corroborate theirs, but in the context of the specific coordination enabled by APIs. For example, the instability of APIs in the MCW team clearly reflects the issue of conflicting requirements.

Managers recognized the coordination effort required to integrate pieces of code provided by different developers. In one instance, for example, the client manager recognized that a developer would not be able to meet his schedule because he had to integrate his source code with source code provided by two other different software developers, members of the same team. This additional effort necessary to integrate different pieces of code has already been identified by Grinter, who called it recomposition work: the work necessary to build a system from its pieces [18]. In another work, Grinter [19] discusses how software architects need to convince other members of the organization to "buy in" their design. The same phenomenon happens in this organization, where software architects bring in to the API design review meeting the client and test teams, which need to approve the API. Indeed, software developers at this organization faced a dilemma: on one hand, they wanted to define APIs early in the process in order to allow independent work, however, on the other hand, they wanted to avoid making the API unstable, which could be avoided only by postponing the definition of this same API. Fowler [11, 14] suggests postponing the definition of APIs to avoid changes and the resulting instability, which does not allow independent work.

In this section, we have contributed an interpretation of how recent research of ours and others in the fields of software engineering and computer-supported cooperative work help explain the issues seen in section 6 and point to encouraging directions. In the next section, we attempt to be even more specific about changes in support tools and practices for alleviating some of these problems.

8. IMPLICATIONS FOR TOOLS

We have discussed how APIs play an important role in the coordination of a collaborative software development effort. Moreover, we discussed how APIs are reification of organizational boundaries in the organization, therefore, at the same time, allowing and constraining collaboration among software developers of different teams. One of the reasons why they hinder collaboration is because they are used in such a way that do not allow software developers' to be aware of their colleagues' actions that might affect their work. Because not all actions are important, we argue that awareness tools need to be able to hide some details from software developers while at the same time, provide important information to let them align their work. This suggests that translucent approaches could be very useful in the design of collaborative software development tools [13]. As mentioned in the previous section, a translucent system makes people's actions visible through intelligent user interfaces and thereby facilitate awareness among collaborators. Similarly, collaborative software development environments need to be able to make software developers' actions visible to the subset of software developers "interested" in these actions. One way of identifying who are those "interested" software developers is by using the concept of APIs. For example, the API consumer needs to be aware of the actions of the corresponding API provider, so that they can both align their work and avoid problems during the integration.

Furthermore, our field study also suggests how to go about deciding which information should be presented and which information should be hidden. Indeed, we argue that the source-code itself contains this information because dependencies among pieces of software create social dependencies among software developers. 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 implemented by *a* and *B* is being implemented by *b*, we similarly find that developer *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* other components. Information from this call-graph combined with authorship information could be used to create a "social call-graph" describing 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 package A to package B, meaning that classes inside package A request services from classes inside package B. Directed edges between authors and packages indicate authorship information. Note that authors are leaves in this graph.

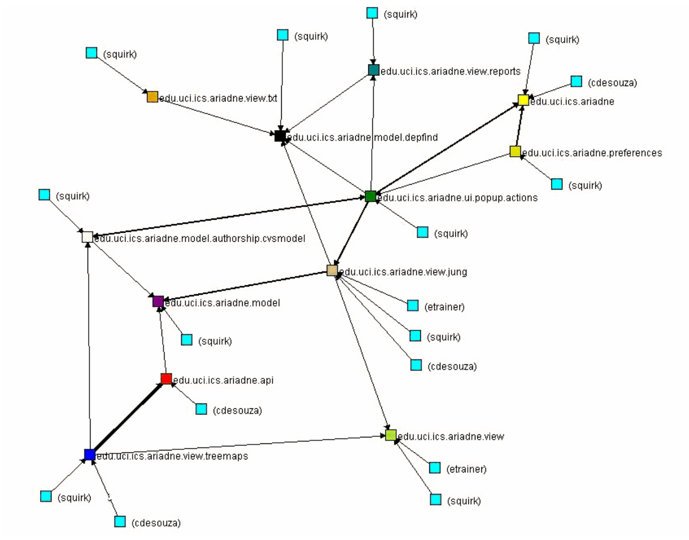
(squirk)
(squirk)
(squirk)
(squirk)
edu.uci.ics.ariadne.view.txt
edu.uci.ics.ariadne.view.reports
(cdesouza)
edu.uci.ics.ariadne
edu.uci.ics.ariadne.model.depfind
edu.uci.ics.ariadne.preferences
(squirk)
(squirk)
edu.uci.ics.ariadne.ui.popup.actions
(squirk)
edu.uci.ics.ariadne.model.authorship.cvsmodel
edu.uci.ics.ariadne.view.jung
(etrainer)
edu.uci.ics.ariadne.model
(squirk)
(squirk)
(cdesouza)
edu.uci.ics.ariadne.api
(cdesouza)
edu.uci.ics.ariadne.view
edu.uci.ics.ariadne.view.treemaps
(etrainer)
(squirk)
(squirk)
(cdesouza)

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

Unfortunately, current software development tools have only focused on the call-graph itself, that is, information from this "social call-graph" has not been explored yet. We argue that this 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, there is no need inform *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 6.3. This approach is very similar to the one adopted in the ExpertiseBrowser system [29], that provides expertise identification based on the number of changes that one committed to the file.

Because of the information that they make have available, "social call-graphs" could easily be transformed in social network graphs, where the relationship among software developers is a dependency relationship. In order to do that, one only needs to get the dependency relationships among components and use them do describe dependency relationships among the authors' of these 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 test 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.

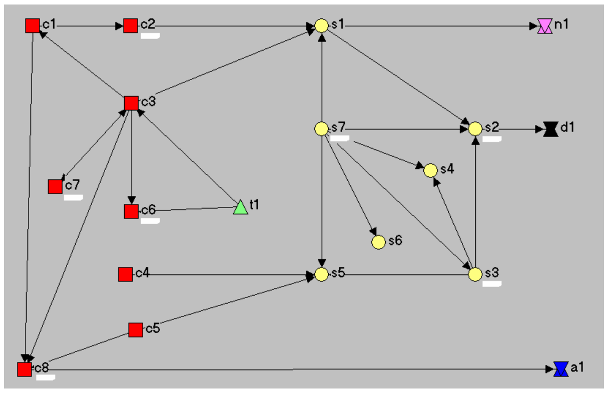

Figure 2: An example of a social network graph describing dependency relationships among software developers

The social-call graph approach could also be used to help support the human-to-human collaboration such as found in agile approaches, like such as Extreme Programming (XP)[3] and Test Driven Development (TDD)[4]. These approaches promote more collaboration through practices like pair programming and stand-up meetings, asking developers to interact and work together more often. For a company that is not using these approaches or that is having problems promoting the technique, the "social-call graph" could be an initial step to facilitate this transition and or adoption. Furthermore, these approaches can benefit from our "social-call graph" because they assume the pre-existence of a closely-knit group of people (or team) that is interacting often. However, this assumption does not always hold, since some groups might be distributed, the project may evolve requiring different kinds of expertise, team requirements may change during prototyping, development, and maintenance phases requiring new people being integrated to the team, and so on. Indeed, recent research in software development teams reveals that there is a large disagreement about team boundaries in distributed and collocated settings [30].

An organization interested in practicing XP, TDD, or other agile approaches as well as other software development organizations may benefit from the "social call-graph", because it can provide valuable information to facilitate these approaches such as:

- which pieces of code will be impacted by one change;
- who are the software developers that need to be notified about this same change;
- who are the most active users of one particular module being refactored;
- who are the most proficient developers of a module being refactored;
- identification the "right" people to complete the missing "expertise" in the group, and so on.

Note that some of the conditions described before, such as distributed software development and different expertise needed are not exclusive of organizations practicing XP, TDD or other agile approaches. Therefore, we believe that other organizations can also benefit from our "social call-graph" approach.

Finally, we argue that APIs should be considered first-level programming constructs, which means that tool support for publication, versioning, diffing, and updating of APIs is necessary. Of course, configuration management tools can easily perform these operations in individual and in set of files. However, these same tools do not provide support for these operations in large set of files, such as APIs. It is important to notice that some open-source projects are beginning to address this problem (e.g., see Jdiff at http://www.jdiff.org/ and DependencyFinder at http://depfind.sourceforge.net/). Furthermore, because of their role as an instrument that facilitates communication among software developers, we also argue that APIs should be given graphical representations, so that they can be designed and discussed and modeled in modeling languages such as UML [34]. Hence, cooperative software development tools could use this information to notify API consumers when changes in APIs were performed, as well as inform API providers about the severity of their changes in the API.

9. CONCLUSIONS AND FUTURE WORK

The notion of APIs is a well known widely used concept in software engineering. APIs are constructs that implement the principle of information hiding, which aims to create well-defined interfaces between two pieces of software to minimize the dependency between them. We performed a field study of a software development project that relied on APIs both for technical design as well as social coordination. We observed several beneficial roles of APIs that confirmed expected uses. Namely, APIs served as contracts among stakeholders, reifications of organizational boundaries, and as a common language among software developers. However, we also observed various limitations of APIs including information overload, instability, integration problems, and lack of awareness. Just as researchers have begun to address the issue of openness with respect to algorithmic aspects of software components, we suggest using concepts of social translucence and awareness to open up software components in a way supportive of collaborative software development.

10. ACKNOWLEDGMENTS

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

11. REFERENCES

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

[2] Barthelmess, P. and Anderson, K. M., "A View of Software Development Environments Based on Activity Theory," in Computer Supported Cooperative Work (CSCW) - Special Issue on Activity Theory and the Practice of Design, vol. 11, 2002, pp. 13-37.

[3] Beck, K., "Extreme Programming Explained: Embrace Change," Addison-Wesley, 1999.

[4] Beck, K., "Test-Driven Development by Example," Addison Wesley, 2003.

[5] 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.

[6] Cheng, L.-T., Hupfer, S., et al., "Jazzing Up Eclipse with Collaborative Tools," in OOPSLA Workshop on Eclipse Technology eXchange. Anaheim, CA, USA, 2003, pp. 45-49.

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

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

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

[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," in Conference on Computer-Supported Cooperative Work (CSCW '92), R. Kraut, Ed. Toronto, Ontario, Canada: ACM Press, 1992, pp. 107-14.

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

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

[15] Fuggetta, A., "Software Processes: A Roadmap," in Future of Software Engineering. Limerick, Ireland, 2000.

[16] Ghezzi, C., Jazayeri, M., et al., "Fundamentals of Software Engineering," Prentice Hall, 1991.

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

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

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

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

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

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

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

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

[25] Kiczales, G., "Beyond the Black Box: Open Implementation," in IEEE Software, vol. 13, 1996, pp. 8-11.

[26] Kiczales, G., Lamping, J., et al., "Open Implementation Design Guidelines," in International Conference on Software

Engineering. Boston, MA, USA: IEEE Press, 1997, pp. 481-490.

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

[28] McCracken, G., "The Long Interview," SAGE Publications, 1988.

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

[30] Mortensen, M. and Hinds, P., "Fuzzy Teams: Boundary Disagreement in Distributed and Collocated Teams," in Distributed Work: New Research on Working across Distance Using Technology, S. Kiesler, Ed.: MIT Press, 2002, pp. 283-308.

[31] Nutt, G. J., "The evolution toward flexible workflow systems," in Distributed Systems Engineering, 1996, pp. 276-294.

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

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

[34] Rumbaugh, J., Jacobson, I., et al., "The Unified Modeling Language Reference Manual." Reading, MA: Addison Wesley Longman, Inc, 1999.

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

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

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