An Empirical Study of Software Developers' Management of Dependencies and Changes

Cleidson R. B. de Souza
Universidade Federal do Pará
Faculdade de Computação
Belém, PA, Brasil
55-91-211-1405
cdesouza@ufpa.br

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

ABSTRACT
Different approaches and tools have been proposed to support change impact analysis, i.e., the identification of the potential consequences of a change, or the estimation of what needs to be modified to accomplish a change. However, just a few empirical studies of software developers' actual change impact analysis approaches have been reported in the literature. To minimize this gap, this paper describes an empirical study of two software development teams. It describes, through the presentation of ethnographic data, the strategies used by software developers to handle the effect of software dependencies and changes in their work. The concept of impact management is proposed as an analytical framework to present these practices and is used to suggest avenues for future research in change impact analysis techniques.

Categories and Subject Descriptors
D.2.9 [**Management**]: Programming Teams; D.2.11; [**Software Architectures**]: Information Hiding; H.5.3 [**Group and Organization Interfaces**]: Computer-supported cooperative work;

General Terms: Human Factors, Experimentation.

Keywords: Change impact analysis, empirical studies, socio-technical aspects, collaborative software development.

1. INTRODUCTION
Change impact analysis consists of a collection of techniques for determining the effects of source-code modifications into software development artifacts [1-3]. Among these approaches, it is possible to include, for example, regression test selection techniques [4] that choose, from an existing test set, tests that are deemed necessary to validate modified software. These techniques have been compared in empirical studies [5]. Another approach adopted by researchers is the construction of software tools to automatically support change impact analysis [2, 3].

In this paper, we take an empirical approach. Instead of focusing on the implementation of new tools or techniques, we studied how and when software developers perform "change impact analysis" in their daily work. Our aim was to identify and analyze software developers' work practices or strategies, and by doing that inform the design of more adequate change impact analysis tools. Our approach is particularly important for software engineering research since, to the best of our knowledge, there are few studies that focus on *how*, *when* and *where* software developers handle the impact of changes in their day-to-day work. Note that it is not possible to study changes in software systems without studying dependencies, since dependencies among software artifacts lead fairly often to additional changes in the software [6]. Therefore, the strategies identified in this paper describe how software developers manage dependencies in general (instead of changes) in software development projects.

Two main related studies were identified. First, Staundemayer [7], who describes the strategies adopted by six different teams to handle dependencies in their work. She describes, for instance, how some teams tried to minimize the influence of external dependencies by assigning developers the primary responsibility of communicating with external developers who provide software components to the team. Staudenmayer also suggests that there is a positive correlation between management of dependencies and performance: among the six teams that she studied in two different organizations, those actively seeking to manage dependencies in their work were the ones with best performance. And second, Grinter [8] who describes strategies adopted by different software development companies. An example is the establishment of "build and release" teams, that need to interact with other teams in the organization to make sure daily builds are provided for those interested. Daily builds allow teams to integrate their work often and consequently to identify problems early in the process.

While Staundemayer focused on *team* strategies, Grinter focused on *organizational* ones, that is, approaches adopted by the organization. While it is true that some of the organizational and team approaches described by these authors are ultimately performed by individuals, the analytical focus of these authors is *not* at the individual level. In other words, most researchers do not focus on the *individual* software developers' strategies to handle dependencies, that is, the work practices they adopt to get their work done and how those practices reflect their concern with the management of dependencies and changes. By not focusing on developers' practices, it is more difficult to understand the weaknesses and strengths of the tools used by software developers[1].

ICSE'08, May 10–18, 2008, Leipzig, Germany.

[1] An exception to this is Grinter's [9] analysis of configuration management tools to support coordination. In contrast, this paper

In order to fully inform the design of more adequate tools that facilitate change impact analysis, this understanding is necessary. That is exactly the focus of this paper. It describes, through the presentation of ethnographic data of two software development teams, the strategies, or work practices, used by software developers to handle the impact of dependencies and changes in their work. As we will illustrate in the paper, their concern with changes is always accompanied by a concern with dependencies. To explicitly indicate software developers' concern with the impact of changes, the concept of impact management is presented. *Impact management is defined as the work performed by software developers to minimize the impact of one's effort on others and, at the same time, the impact of others into one's own effort.* In addition, this paper discusses how software engineers use their knowledge about the dependencies in the software architecture to properly perform impact management and how change impact analysis tools could be extended to reflect the results identified in the empirical study.

The rest of the paper is organized as follows. Section 2 describes the research sites studied, MVP and MCW, as well as the methods used to collect and analyze data from these sites. After that, section 3 discusses the concept of impact management. Then, section 4 presents the ethnographic data of the MCW and MVP teams. Section 5 discusses the observations from these teams. Finally, section 6 presents the final comments and future work.

2. RESEARCH SITES AND METHODS

To identify and understand the strategies used by software developers to handle software dependencies, we conducted two qualitative studies at different large software development organizations. The first field study was conducted during the summer of 2002, when the first author was an intern in the MVP team, and the second one was performed during the summer of 2003. We adopted participant and non-participant observation [10] and semi-structured interviews [11] for data collection. Data analysis was conducted by using grounded theory techniques [12]. Details about each team as well as the methods used are described below. For confidentiality reasons, the names of the teams and organizations are not their real names.

2.1 MVP

The first team studied develops a software application called MVP, a nine-year old software composed of ten different tools in approximately one million lines of C and C++ code. Each one of these tools uses a specific set of "processes." A process for the MVP team is a program that runs with the appropriate run-time options and it is not formally related with the concept of processes in operating systems and/or distributed systems. Running a tool means running the processes required by this tool with their appropriate run-time options.

The software development team is divided into two groups: the verification and validation (V&V) staff and the developers. The developers are responsible for writing new code, for bug fixing, and adding new features. This group is composed of 25 members, three of whom are also researchers that write their own code to explore new ideas. This group is spread out into several offices across two floors in the same building. V&V members are responsible for testing and reporting bugs identified in the MVP software, keeping a running version of the software for demonstration purposes and for maintaining the documentation (mainly user manuals) of the software. This group is composed of 6 members. Developers and V & V team members share offices in the same building.

The MVP group adopts a formal software development process that prescribes the steps to be performed by the developers. For example, all developers, after finishing the implementation of a change, should integrate their code with the main baseline. In addition, each developer is responsible for testing its code to guarantee that when he integrates his changes, he will not insert bugs in the software, or, "break the code" [13]. Another part of the process prescribes that, after checking-in files in the repository, a developer must send e-mail to the software development mailing list describing the problem report associated with the changes, the files that were changed, the branch where the check-in will be performed among other pieces of information.

The first author spent eight weeks as a member of the MVP team. He made observations and collected information about several aspects of the team, talking with colleagues to learn more about their work. Additional material was collected by reading manuals of the MVP tools, manuals of the software development tools, formal documents (such as the description of the software development process and the ISO 9001 procedures), training documentation for new developers, problem reports, and so on. All MVP team members agreed with the data collection. Furthermore, some of the team members agreed to be shadowed for a few days. These team members belonged to different groups and played diverse roles in the MVP team. They worked with different MVP processes and tools and had different experience in software development, which allowed us to get a broad overview of the work being performed at the site. Eight MVP team members were interviewed during 45 to 120 minutes according to their availability. To summarize, the data collected consist of a set of notes that resulted from conversations and documents, and observations are based on shadowing developers.

This dataset was analyzed using grounded theory techniques [12]. The results of this analysis pointed out the importance of the dependencies between the software development artifacts and the practices adopted by software developers to deal with these dependencies, in particular how changes in an artifact impacted other artifacts. Therefore, the second data collection in the MCW team focused on these same aspects.

2.2 MCW

The second field study was conducted in a software development company named BSC. The project studied, called MCW, is responsible for developing a client-server application. 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 is comprised of the project lead, development manager, user interface designers, and so on. The client team is developing the client side of the application, while the server team is developing the server aspects of the application. The infrastructure team is working in the shared components to be used by both the client and server teams. Finally, the test team is responsible for the quality assurance of the product, testing the software produced by the other teams.

---

describes individual practices used by software developers while they use the several tools that they use to perform their work.

The MCW project 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 / team) 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 this project interact with software developers in other parts of the organization.

BSC enforced the adoption of a *reference architecture* during the development of software applications. The BSC reference architecture prescribed the adoption of some particular design patterns [14], but at the same time it gave software architects across the organization flexibility in their designs. This architecture is based on tiers (or layers) so that components in one tier can request services only to the components in the tier immediately below them [15, 16]. Data exchange between tiers is possible through well-defined objects called "value objects." Meanwhile, service requests between tiers are possible through Application Programming Interfaces (APIs) that hide the details of how those services are performed (e.g., either remotely or locally, with cached data or not, etc.). In this organization, APIs were designed by software architects in a technical process that involved the definition of classes, method signatures, and other programming language concepts, and the associated documentation. APIs were both a technical construct and a organizational mechanism to separate teams allowing them to work independently [17].

Regarding the data collection, in this field study, we adopted non-participant observation [10] and semi-structured interviews [11], which involved the first author spending 11 weeks at the field site. Among other documents, meeting invitations, product requests for software changes, emails and instant messages exchanged among the software engineers were collected. All this information was used in addition to field notes generated by the observations and interviews. We conducted a total of 15 semi-structured interviews with members of all five sub-teams. Interviews lasted between 35 and 90 minutes. An interview guide was, to some extent, reused from the MVP field study to guarantee that similar issues were addressed. This data was analyzed using grounded theory [12] to understand the role of APIs in the coordination of MCW developers and is reported in [17].

2.3 Data Analysis

After the second data collection, datasets from the two different teams were integrated into a software tool for qualitative data analysis, MaxQDA2. After that, the data collected was jointly analyzed by using grounded theory [12],. This technique does not require a prior theory about the data, that is, a set of hypothesis to be tested. Instead, the goal of grounded theory is precisely to generate theory grounded exclusively on the existing data. In other words, it aims to develop a theory or explanation about what is going on in the field, or more specifically, what is available in the data collected.

Grounded theory proposes three major steps. The first step is called *open coding*, in which data (in this case, interviews and field notes) are micro-analyzed (line-by-line) to identify categories. Categories are grouping concepts put together under a more abstract high-order concept to explain what is going on [12, pg. 113]. Categories are created to minimize the number of elements that the researcher needs to consider. During the next step, *axial coding*, categories were broken into subcategories: whereas categories stand for phenomena, subcategories answer questions about the phenomenon, such as when, where, why, who, how, and with what consequences [12, pg. 125]. During axial coding, one identifies properties and dimensions of each one of the categories. For instance, a category identified in our data is called impact and it is used to model the work that a developer needs to perform *because of* the work that another developer performed. In this case, one important attribute of this category is its degree, meaning the intensity of an impact on one person's work, or the amount of re-work that the person will need to perform. Finally, during the last step, *selective coding*, the most important categories are selected to be core categories, that is, the categories that will be used to describe the emerging theory. Other categories are then associated with the core categories. In this paper, impact becomes one of the central categories. In contrast, APIs are less central and will be used to illustrate a technical approach used by software developers to reduce the impact on each other's work As a result of the usage of grounded theory techniques, the impact management framework emerged as a useful framework to explain what we observed in the datasets. This framework is explained in the following section.

3. IMPACT MANAGEMENT

One of the reasons why software development is difficult is the intricate web of dependencies among artifacts and software development activities [18]. The work required to manage these dependencies can be seen as impact management. Impact management is defined as the work performed by software developers to minimize the impact of one's effort on others and, at the same time, the impact of others into one's own effort. *Viewing dependency management as impact management draws attention to the developers' concern about being impacted by changes made by and impacting their colleagues during software development efforts.* It illustrates how one orients himself toward his colleagues so that both can get their work done.

There are three main aspects to impact management. First, finding the network of people that might affect one's work and that might be affected by one's work, a concept that we call *impact network*. This network is influenced by several factors, but as we will discuss, mainly by the software architecture. Identifying the impact network is the most important aspect of impact management, and in fact our data suggest that when the impact network cannot be properly established, software development activities become much more complex.

Second, *forward impact management* is the work to assess the impact of one's own work on one's respective impact network and inform them of such impact. Examples of these practices include filling the fields of problem reports in the bug-tracking tools, sending emails with notifications of changes, code reviews that aim to understand the impact of the changes in the software architecture, and so on

Finally, *backward impact management* consists of assessing the impact of the work performed by developers in one's impact network on one's own work, and the appropriate actions to avoid such impact. Software developers adopt several approaches to avoid such impact, such as "back merges" that incorporate changes from others into one's workspace, attending meetings with other teams, and so on. Again, the MVP and MCW practices are illustrated in the next section. As expected, as much as possible, software developers try to anticipate changes in the software system so that they can prepare themselves for these changes. Figure 1 below illustrates the concept of impact management: the impact network and the backward and forward impact management aspects.

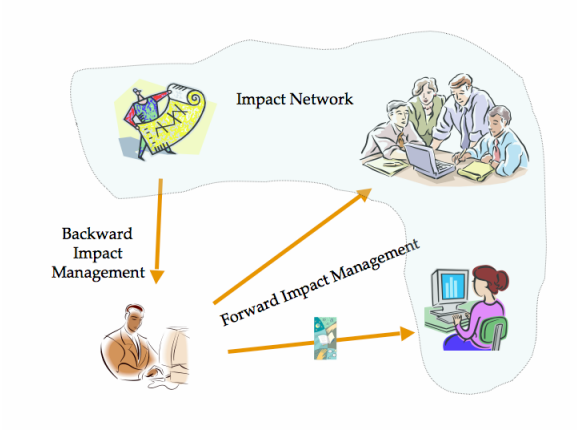

Figure 1 - Impact Management

Note that the distinction among backward and forward impact management is analytical: they are iterative, interwoven among themselves and among other developers' practices. Furthermore, these aspects are also complementary: whenever a developer is performing forward management, he is facilitating the backward management to be done by others, and vice versa. For instance, whenever a MVP developer decides to postpone his or her check-in's, he or she is allowing other developers not to worry about deciding whether to recompile their code.

The focus of this paper is on the strategies of the MVP and MCW teams. Whenever a MCW team member decides to attend meetings of another team, he or she is doing backward management. This team member is doing extra work to understand how this other team's changes can have an effect on him. Since the focus is on these teams, we are not classifying the other team's work of allowing an external developer to attend a meeting as forward impact management. That being said, some strategies to be described embody forward and backward management concerns: they are complementary aspects of the same coordinative practices. Other strategies are specific (either backward or forward) and do not have a clear counterpart.

Impact management strategies or practices can be understood at different levels of analysis: Some are performed by individuals, whereas others are adopted by the entire team, and finally some of them are organizational mandates. Organizational strategies are those mandated by the organization, namely, the implementation of the reference architecture and the usage of application programming interfaces. Team strategies are those mandated by managers, usually described in the software development process. Finally, individual strategies are mundane practices performed by software developers, the "ordinary methods that ordinary people use to realize their ordinary actions" [19, pg. 29].

4. IMPACT MANAGEMENT IN THE MVP AND MCW TEAMS

In the following sections, we will describe the different aspects of impact management by presenting the practices performed by each team. We will describe just a few of these practices, because of space limitations. More details can be found at [20].

4.1 The Practices of the MVP Team

4.1.1 Impact Network Identification

MVP developers did not face problems identifying their impact network due to several factors. Most developers had been working in the project for a couple of years and had learned about which developers work in which parts of the source code. In addition, the software development process prescribed that software developers should send email to all other developers before integrating their changes in the shared repository. These emails are broadcast to the entire set of software developers and V&V personnel, who then uses this information to find out whether the new changes in the code affect their work or not. Especially because of this step in the MVP process, developers did not need to worry about identifying their impact network: they would always inform and be informed about changes in the code.

Broadcasting emails was also useful because of the lack of modularity of the MVP software: a change in one particular "process" could impact all other "processes." According to a senior MVP developer:

"There are a lot of unstated design rules about what goes where and how you implement a new functionality, and [where] *it should be in the* [software architecture]. *Sometimes you can almost put functions anywhere. Every process knows about everything, so just by makefiles and stuff you can start to move files where they shouldn't be, and over time it would just become completely unmaintainable.…yeah, every process talks to every other one in a way and there are some basic architecture issues …"*[Emphasis added]

The above quote suggests that, because of the structure of the software architecture, the impact network could be as large as the entire team of developers, which would make it necessary to broadcast changes to all developers. However, MVP team members sometimes needed more contextualized information about their impact network. This was also accomplished by using information available on these emails. They associated the author of the emails describing the changes with the "process" where the changes were occurring: if one developer repeatedly performs check-ins in a specific process, it is very likely that he is an expert on that process. Therefore, if another developer needs help with that process he will know who to contact for help. This strategy was reported by a developer, who had been working for only two years in the MVP project. Knowledge about which developer works in each "process" is important for this developer and other newcomers because all MVP developers might need to make changes in "processes" in which they are not experts, a not unusual situation. In order to do that, MVP developers would often contact the process owner to learn about the process. After finishing their changes, MVP developers will again contact the owners of the process to verify whether their changes in the process are going to impact the work of these process owners.

4.1.2 Forward Impact Management

The MVP team adopts a software development process that prescribes that after checking-in code in the repository, a developer needs to send an email about the new changes to the software developers' mailing list. However, we found out that MVP developers send this email before their check-ins. More importantly, MVP developers add to this email a brief description of the impact that their work (changes) will have on others' work. The following list presents some comments sent by MVP developers:

"No one should notice."

"[description of the change]*: only* [tool1] *users will notice any change."*

"Will be removing the following files. No effect on recompiling."

"Also, if you recompile your views today you will need to start your own [process1] *daemon to run with live data."*

"The changes only affect [y] *mode so you shouldn't notice anything."*

"If you are planning on recompiling your view this evening and running a MVP tool with live [type of data] *data you will need to run your own* [process1] *daemon."*

Details of this approach, however, were left to be decided by the developer sending email. According to MVP Developer-02:

"sometimes when, if the thing is going to have wide impact then its up to the person who wrote that email to say that this is going to have wide impact and everyone is going to have to recompile. Like if [developer's name] *changes the format of the* [process] *file, something that needs to be read in, he upgrades the* [component] *or something like that. That means that old files may no longer be useful and the files are going to have – people have collections of these files that are going to have to be converted somehow."*

This extra-step, not defined in the software development process, taken by MVP developers clearly illustrates the idea of impact management. MVP developers try to anticipate the impact of their work in their colleagues and even suggest courses of action to be taken to minimize this impact.

In addition to adding impact descriptions to their emails, MVP developers often postponed their check-ins in parts of the code until later in the day so that the other developers did not waste their time with recompilation. This was only possible because developers were aware that some parts of the MVP software affected all other parts of the system, so that changes in them would require most other developers to recompile their code, a time-consuming process. According to MVP Developer-04:

"People also know that if they are going to check-in a file they do it later in the afternoon. A lot of times you will find that people they do a lot of work, they are aware and most of us are here during the day and if you are going to do a check-in and it's going to cause anybody who recompiles that day to watch their computer for forty five minutes, I mean most times you see that come in at two or three in the afternoon … you don't see someone doing a [change in a global file] *check-in at eight in the morning because everybody all day is going to have to sit there and recompile … Most of the time you see that stuff in the afternoon … a common courtesy …"*

This illustrates, again, how MVP developers engage in forward impact management: by postponing their check-ins, they minimize the impact they inflict on their colleagues. More importantly, however, *they engage in this work because they are aware of the details of the software architecture, and use this knowledge to adjust their work accordingly*.

4.1.3 Backward Impact Management

Software developers use the emails exchanged among them to find out if they have been engaged in parallel development. Parallel development happens when more than one developer needs to make changes in the same file. This means that the same file is checked-out by different developers and all of them are making changes in the different copies of this file in their respective workspaces [21]. If a developer is engaged in parallel development with other developers and they checked-in their changes in the main branch before he or she did, that developer will have received emails from the other developers about their check-ins. By reading these emails, the developer will be aware that he is engaged in parallel development because these emails need to describe, among other things, the files that have been checked-in. In this case, this developer will need to perform an operation known in the MVP team as a "back merge." This operation is supported by the CM tool adopted by the team and is required before this developer can merge his or her code into the main branch.

Parts of the MVP software contain important definitions that are used throughout the whole program. This means that they are constantly changed by several developers in parallel; back merges thus are performed fairly often:

"It depends on … there are certain files, like if I am in [process1] *and just in the* [process2] *that* [back merges] *is probably not going to happen, if I am in the* [process3] *there is like … there is socket related files and stuff like that. I think* [filename] *and things of that sort. There's a lot of people in there. The probability of doing back merging there is a lot higher. What I will probably try to do is discard my modifications and or I'll save my modifications and then, right now I'll see if I can put myself on top of it because at that point there's stuff supposedly already committed so there's nothing I can do except build on top of them."*

In other words, the MVP software architecture plays a major role in allowing parallel development. This situation in itself is not problematic because CM tools, most of the time, can handle most of the situations involving back merges. However, parallel development affects the testing activities that need to be repeated to guarantee that one's changes do not interact in unexpected ways with the changes that had been checked-in before. As indicated by MVP developer-04:

"When I check-in … at that stage in the game I am looking at how many people are sitting here using it and also has anybody checked it in since I checked it out? 'Cause if that is the case, then I certainly need to get those changes and test again before I check back in. So if three or four people have done work and checked a thing in since I checked the file out and now I am ready to check in and I have tested all of my changes, well I need to retest all my changes with the three or four different people who have checked in since then to make sure what they have done doesn't change something that I've done."

Another developer reported that he even tries to "speed-up" his work to finish it sooner and avoid back merging and consequently new testing activities. In contrast, to avoid back merges without avoiding parallel development, MVP developers perform "partial check-ins." In a partial check-in, a developer checks-in some of the files back in the main repository, even when he has not yet finished all the changes required for the bug fix or new functionality. To avoid problems, this partial check-in can not break the build. This strategy reduces the number of back merges needed and minimizes the likelihood of conflicting changes during parallel development. In other words, MVP developers employ partial check-ins to avoid being affected by other developer's changes in the same files. In short, a partial check-in is a form of backward impact management.

4.2 The Practices of the MCW Team

4.2.1 Impact Network Identification

In contrast to MVP developers, MCW developers faced several problems while identifying their impact network. In fact, during interviews with MCW developers, we found out that many of them were not aware of their network: MCW server developers did not know who was consuming the services provided by their components, and MCW client developers did not know who was implementing the component that they depended on. Because they were not aware of their impact network, developers did not receive important notifications (e.g., important meetings they needed to attend) and faced other problems.

To deal with the problem of finding their impact networks, developers relied on their personal social network. A developer, for instance, reported talking to up to fifteen people before finding the right person:

MCW Developer-15: "… I am kind of merciless in trying to find the right person. I have shotgunned up to four or five people at once to say 'do you know who is responsible for this?' and then gotten some leads and followed up on those leads and talked to as many as ten to fifteen different people."

Managers also played an important role in this process because of their larger social network. MCW developers would contact their managers so that these managers could identify the person they wanted to find.

Another way of bringing MCW developers together was the API design review meetings. Within the MCW team, these meetings were scheduled to discuss the APIs being developed by the server team. The following people were invited: API consumers, API producers, and the test team that eventually would test the software component's functionality through this API. In addition to guaranteeing that the API met the requirements of the client team and that this team understood how to use it, this meeting also allowed software developers to meet. After that, the server team provided APIs to the client team with "dummy implementations" to temporarily reduce communication needs between them, thus allowing independent work. Unfortunately, time would pass between these meetings and the actual implementation of the API. In the meantime, changes in developers' assignments would cause communication problems because developers did not know about each other anymore. In short, changes in assignments changed the impact network, thereby making the work of software developers more difficult to be coordinated.

Software developers acknowledged that this situation was problematic. For instance, a developer suggested that a database containing information about who was doing what in the team was necessary: "sometimes you wanna talk to a developer … the developer in the team who is working in this feature [that you need]." Actually, identifying the impact network was a problem in the entire BSC organization. Indeed, BSC managers created yet another discussion database that developers could use to find out who were the people necessary to answer their questions. Because of the large number of databases already in use by the BSC teams, managers had to slowly convince BSC developers of the importance of this particular database:

"The management team is really trying to socialize the idea that that [the discussion database] *is the place to go when you have a question and you don't know who can answer it. They are really trying to socialize that people should give a scan to it every once in a while to see if they can help and answer a question. The amount of traffic there has picked up quite a bit in the last couple of months, especially in the past couple of weeks. My team has not gotten that message a hundred percent yet. There is a tool for it and a place to go that I have had a lot of success with when I use it; it is just that the message has not gotten out yet that that is the place to go. One of the things that happens when you have so many databases* [is] *it takes a while for one to emerge as the place to be. This is turning out to be the place to be."*

Similar to the MVP team, the software architecture also influenced the process of impact network identification. Applications developed in the organization should be designed according to a *reference architecture* based on layers, so that components in one layer could request services only to components in the layers immediately below them. In other words, each layer had specific dependencies: if it was changed, it would be necessary to inform only the developers dependent on this layer, instead of broadcasting changes to all developers. In fact, interviewees clearly suggested being interested in changes only in particular parts of the architecture, parts that impacted them.

Another important factor that influenced the impact network identification was an organizational one, the large-scale reuse program adopted by BSC. This program, as discussed before, led MCW developers to interact with developers in different teams who could be located in other parts of the country and even in other countries. This was necessary to allow components to be reused in the organization. In fact, a developer complained about the need to simplify the "communication channels" in the organization to avoid having to interact with different managers to find out who was the person responsible for implementing a particular component or its services. This same developer reported that one of the teams providing a component to his team was not even aware of his team's need. On another occasion, a developer tried to find out whether she could use a particular user-interface (UI) component. The UI designer working with her indicated a developer in Japan who was using this same component. It was this Japanese developer who recommended to her another developer, now in the U.S., who was implementing the UI component she wanted!

This problem was aggravated by the young age of the project, according to [MCW Developer-15]:

"When you sit on a team for two years you know who everybody is. Even peripherally you know who people are. So if we had to get answers about [another BSC product in the market for years] *we have so many people on the team who were on the team for so long* [a] *period of time they can get the answer immediately. They know who the person is even if they have never met them. We don't have that in this group because it takes time for those relationships to develop, for there to be … enough of the investigations … like I talked to so and so and talked to so and so and so on. You only have to go through that once or twice because once you have gone through that you know the person. I think part of that frustration is how you spin up those relationships more quickly. I don't know if you realize this but this team has only been in existence since last year. So it is a ten-month-old team."*

4.2.2 Forward Impact Management

MCW developers had an expectation that major changes in the code should be accompanied by notifications so that the build did not break. In fact, developers reported contacting their colleagues in different occasions to warn them of major changes in the code and

their associated implications. For instance, group meetings provided an opportunity to developers to inform their colleagues of changes that they would make that would impact the others. Developers also informed their colleagues on other teams. For instance, developers informed the installation team of new files being added or removed so that the installation procedures could be updated with this information. In other cases, server developers negotiated with client developers changes in one of the APIs that existed between the teams before actually performing the changes.

However, the usefulness of these notifications was contingent on knowing who to contact. As discussed in section 5.3, not all MCW developers were aware of their impact network; therefore, they were not able to provide important notifications.

Similarly, three different developers reported interest in being able to allow developers to inform colleagues of "critical updates" in the code that need to be picked up by all other developers using the code. For example, in the quote below MCW Developer-05 argues that he would like to be able to flag a file, and by doing that, to inform his colleagues that a new version of that file would be checked-in the repository pretty soon:

"If the developers can say … this is the latest file, something that says that the other files that other people have are old … and if this person only gets this one file and he may update and other people should get that latest one, if there is some way to mark that and it reflects on everyone, then you should get the new file which would be this one, I think that everyone should get this one. People wouldn't be developing against this old file."

On the other hand, developers are concerned about receiving too many notifications of things that were not relevant to them. That is, they were concerned about not being in one's impact network and still receiving notifications. This occurred because of the size and number of discussion databases being adopted in the organization. In fact, this was a common problem reported by MCW developers.

Notifications are useful only to some extent: once an API is made public to the entire organization or even external customers, control of who is using it is lost, and therefore notifications are not necessary anymore because these APIs cannot change anymore. Using the terminology proposed, the *impact network* became unknown. As described by a server developer:

"We have latitude to change it [the API] *as long as we are talking about an unpublished or semi-private API. If it is a contract between us and the client people, we probably have more latitude to change it and therefore they can trust it a little less than if it was a published API. At that point it would be very difficult to change it because people would be relying on … right now we control everything that has a dependency, we control all the dependencies because the only people who are using the API are our own client teams and test teams and we can negotiate changes much easier than if they were external customers that were unknown to us or people in the outside world who we don't control and who also could not readily change their code to accommodate our API changes. We would have to go about carefully deprecating, evolving, ah, end of life some features."*

4.2.3 Backward Impact Management

Backward impact management was particularly important to MCW developers because of the several dependencies that they had in teams spread in the whole organization. We will call these "external dependencies," to contrast with dependencies among the sub-teams of the MCW team. External dependencies are especially important, given the organization-wide reuse program adopted by BSC.

The first and perhaps more important strategy used by MCW developers was *to use other teams' APIs*, which was in accordance with the BSC reference architecture and best practices in software engineering. By implementing against components' interfaces, developers are minimizing the impact that changes in these components could have on their own work. Developers indeed took extra steps to guarantee that they would use other components' APIs. For instance, a developer contacted another team to request them to extend their APIs in order to provide the services he required. Because of the different schedules being followed by BSC teams, sometimes MCW developers would even volunteer to make the changes in the code. This was especially useful when other teams were on a tight schedule for new releases of their products.

Adopting APIs was a design decision taken by BSC software architects. However, software developers were the ones who actually talked to their colleagues and negotiated the exposition and/or implementation of new services in an API. In other words, although APIs were organizational practices to deal with software dependencies, the communication, coordination, and negotiation to make them useful were actually performed by software developers.

In addition, we interviewed more than one *developer who had switched teams to implement the services required by his own team*. One developer, for instance, had moved to another team that provided user interface components to the MCW team, with the unique goal of implementing the UI components needed by MCW client developers. He acted as the "contact person" and forwarded information about the new team that was useful for the MCW team, and vice-versa.

In another occasion, a client developer "followed" his dependency in order to switch teams: he had a dependency on the server team, who actually had a dependency in the infra-structure team, who depended on an external team to provide the component. To simplify the communication channels and make sure that the client team would have the component when needed, the manager of the client team decided to "lend" this developer to this external team. *This example also illustrates how the entire software architecture influenced the coordination of the work. It is not only the "direct" dependencies that affect a software developer's work, but also "indirect" dependencies because those influence the direct dependencies.*

Obviously, whenever possible, developers would contact their API providers to get information about the status of their work. Others would even *attend these teams' group meetings* to learn about these teams' plans and consequently assess how these plans would affect their own work. The information that these developers would get with their colleagues (whom they depended on) was shared with the other team members during MCW client or server team meetings. Another strategy adopted by the MCW team was to "group" requirements to make sure that all inputs were listened to and therefore the design would not have to change later to reflect other needs. These practices were adopted by both MCW client and server developers.

In general, MCW developers tried to anticipate changes in the code – that is, they tried to be aware of changes in the code they depended on, so that they could plan their own work accordingly. Again, this was only possible once MCW developers found out who was in their impact network.

4.3 Additional Practices of the MVP and MCW Teams

Table 1 presents a summary of the strategies that we identified. The detailed description of all strategies can be found in [20].

5. DISCUSSION

It is interesting to note that the diverse strategies described in the Table 1. What is more interesting, however, is to note that these practices address a common purpose: to minimize the impact of one's effort on others, ad the impact of others into one's own effort. This suggests that these practices can perhaps be substituted for one another, or that there are situations in which some of them might be more appropriate than others. Furthermore, a team in another organization could even adopt a different set of practices, but we argue that it would be possible to identify the same rationale for their actions.

5.1 The fluidity of Impact Networks

As illustrated by ethnographic data in the previous section, the impact network of MVP developers changes during the task of performing a change in the software. Temporarily, the impact network consists of the "process" owner, who is the person more likely to be impacted by the developer's modifications.

Team	Strategy	Aspect
MVP	Learning from Email Notifications	Impact Network
MCW	API Design Review Meetings	Impact Network
MCW	Personal Network	Impact Network
MVP	Sending Email Notifications	Forward
MVP	Reading Email Notifications	Backward
MVP	Impact Descriptions	Forward
MVP	Error-Checking	Backward and Forward
MCW	The Reference Architecture and APIs	Backward and Forward
MCW	Pre-Testing Activities	Backward and Forward
MCW	Build Document	Forward
MVP	Back Merges	Backward
MVP	Partial Check-ins	Backward
MCW	Handling External Dependencies – APIs and adaptors	Backward
MCW	Handling External Dependencies – "Exporting" developers	Backward
MCW	Handling External Dependencies – Being aware by attending meetings, engaging in communication, and so on.	Backward
MCW	Handling External Dependencies – Group requirements	Backward
MVP	Problem Reports	Forward
MVP	Formal Code Reviews	Forward
MVP	Informal Code Reviews	Forward
MVP	Holding onto Check-ins	Forward
MCW	Notifications	Forward

Table 1 - Summary of Strategies

Once the changes in the code are finished and are ready to be checked-in in the main repository, the impact network changes to the entire MVP team. The impact network of developers engaged in parallel development also includes the other developers who are changing the same files. In this situation, to minimize the size of the network, developers perform partial check-ins of files that are more likely to lead to parallel development. Changes in software developers' impact network also happen in the MCW project, where changes in assignments lead to changes in the impact network. The frequency of these changes is very different in these two settings: MVP changes occur during the course of work on a bug fixing, while MCW changes are slower, but permanent. The fluidity of the impact networks suggests that impact analysis tools can be even more useful when they are integrated into the software development process, so that they can adjust their results according to the context at hand. In addition, these tools should take into account not only the current changes of a developer, but also the changes his/her colleagues are performing [22, 23].

5.2 Impact Networks and Roles

The results of this study also suggest that different impact analysis approaches are necessary to different types of users, according to their roles in the software development process. For instance, MVP process owners are interested in evaluating whether and how the changes affect the software architecture. That is, the concern is with the source code being changed, but as this code is represented at the architectural level. In this case, approaches like software reflexion models [24] seem more appropriate. Meanwhile, the developer responsible for the documentation needs to evaluate the change itself to determine whether the MVP manuals need to be changed as well. That is, at the same time that the changes to be used in the change impact analysis tools limited to *changes in the source code*, the changes also included the *information that exists in the problem reports (PRs) of the MVP project*, since there is a field in the PR that developers fill to indicate the need to update the manuals.

5.2 The Size of the Impact Network

Another important aspect is the size of the impact network, the set of software developers being impacted or impacting a specific developer. For instance, before an API was made available to other developers or teams, it could be changed without problems because no one would be affected. When this API was available to a small number of teams, changes had to be negotiated to minimize their impact on these teams. Finally, whenever the API was made available to external customers, it could not be easily changed. These "public" APIs had to go through a slow process of change in which some methods were slowly marked to indicate that API consumers should stop using them and start using the new methods. In other words, when the impact network expanded, the practices adopted by the software developers had to change to accommodate the larger number of colleagues being impacted.

In other words, our results suggest that software developers of the MVP and MCW teams are not only concerned with the impact of the changes themselves, but also with the number of clients (and their importance to their business) that will be affected by such changes. As the MCW data suggests, it might not even be worthy to perform some changes when the number of clients affected – the impact network – is too large. Based on these observations, we suggest that current change impact analysis tools should take into account the number of affected clients while determining the effect of a change. It is not only important to choose which tests are necessary to validate a modified software [4], but also to evaluate

the number of clients of that software that will be required to re-execute their integration tests.

5.3 Partial Check-ins

One important strategy used by software developers was the partial check-in. According to this strategy, software developers checked-in code that was partially finished in files heavily dependent upon (and consequently with a high degree of parallel development [21]). This practice suggests that change impact analysis tools need to be able to work with changes that have already been integrated to the main repository. That is, instead of only analyzing changes in software developers' workspace, tools need to be able to allow software developers to include changes already in the repository. Furthermore, it is also necessary to consider the higher likelihood of existence of errors in these files [21] when analyzing the impact of the developers changes.

5.4 On the Impact Management Strategies

Impact management involves two aspects: on the one hand, software developers employ strategies to minimize the impact of their colleagues' actions into their work, and on the other hand, software developers employ strategies to minimize the implications of their work into their colleagues'. These aspects are *complementary*: whenever a developer tries to minimize the impact of his work into others' work, he is also reducing the work done by others to minimize the impact of his work into their own work.

An interesting result from this field study is the observation that software developers in these teams are aware of components dependencies, and, more importantly, that they make use of this information to guarantee the smooth flow of work. For instance, a MVP developer only chooses to hold-on to his check-in because he is aware that a particular component will cause a lot of impact in their colleagues, since this component is a major source of dependencies in the software architecture. Similarly, a MCW developer switched from team A to team C because he knew that according to the MCW software architecture, his team (A) depended on a component implemented by team B, and that B's component was dependent on a component implemented by team C. This developer "followed" the software architecture to find out to which team he should switch. In other words, software components with a higher degree of dependencies had to be handled more carefully because of their larger "potential" for impacting other components in the architecture. The number of dependencies of a software component influences software developers' activities toward this component, i.e., software developers *do know* the degree of dependency in parts of the MVP software and use this knowledge. Accordingly, change impact analysis tools could leverage software developers' effort and knowledge to "guide" their focus, execution and results. Current approaches are mostly automatic limiting the amount of user intervention.

5.5 Limitations of Current Tools

One aspect that largely influenced impact management was the flexibility of the collaborative and software development tools used. The tools used by MVP and MCW developers are described in Table 2 below with an indication of which strategy they supported. Overall, all the tools used were flexible enough to support the impact management practices adopted by these developers. This is not surprising since these tools are indeed built to be flexible and usable in different settings.

Tool	**Strategy**
Email	Learning from Email Notifications, Personal Network, Sending Email Notifications, Reading Email Notifications, Impact Descriptions, API Design Review Meetings (sending invitations), Formal Code Reviews, Notifications
Instant Messenger	Personal Network, Handling External Dependencies – Being aware by attending meetings, engaging in communication and so on
Configuration management	Error-Checking, Back Merges, Partial Check-ins, Formal Code Reviews, Informal Code Reviews, Holding onto Check-ins
Bug-Tracking	Problem Reports
Integrated Development Environment (IDE)	Error-Checking, The Reference Rrchitecture and APIs, Pre-Testing Activities, Handling External Dependencies – APIs and Adaptors
Text Processor	Building Document, Handling External Dependencies – Group requirements

Table 2 – Tool Support for Impact Management Strategies

In this case, it is more important to analyze the tools according to the analytical framework of impact management that we proposed [25]. According to this framework, developers perform extra work to minimize impact from and to their colleagues' work. However, as illustrated in the previous sections, this is better achieved when these developers are aware of their impact network. The question that arises, then, is: how do these tools support impact network identification? Email along with configuration management and bug-tracking tools can potentially allow this activity. However, organizational factors come into place interacting with the tool usage and making this process much more difficult. In addition, these tools do not leverage information about the software dependencies to facilitate software developers' work. A similar conclusion can be reached about the tools and their support to forward and backward impact management: they are flexible enough to support the practices, but they do not have (or leverage) knowledge about the software dependencies, and as we have illustrated, software developers use this knowledge to decide what to do next.

6. FINAL REMARKS

Change impact analysis techniques are an important aspect of software evolution [6]. Accordingly, different approaches and tools have been proposed. Few empirical studies can be found describing how software developers perform change impact analysis in their daily. Furthermore, these studies focus on *organizational and team* approaches, instead of focusing on *individual* approaches. That is the contribution of this paper: it described an empirical study of two software development teams, detailing the strategies used by software developers to handle the effect of software dependencies and changes in their day to day work. In addition, the concept of impact management was used to explain the meaning of these practices and to evaluate how current tools support impact (change) management. Finally, the results of the empirical study were used to suggest improvements to change impact analysis and software engineering tools in general.

Currently, we are building a change impact analysis tool that aims to support the impact management framework and its identified features (fluidity and variable size of impact networks, etc). That is, instead of focusing on particular strategies, which seem organization-specific, our tool will support the common purpose of minimizing impact among software developers. Impact in this case is a socio-technical aspect, involving both the artifacts to be changed as well as the other developers involved in these changes.

ACKNOWLEDGMENTS

This research was supported by the Brazilian Government under grants CAPES BEX 1312/99-5 and CNPq 479206/2006-6, by the U.S. National Science Foundation under grants 0534775 and 0205724, and by an IBM Eclipse Technology Exchange grant and a Microsoft grant. We would like to thank the reviewers for their insightful comments and suggestions.

7. REFERENCES

[1] Bohner, S.A. and Arnold, R.S. Software Change Impact Analysis. IEEE Computer Society, 1996.

[2] Ren, X., Shah, F., Tip, F., Ryder, B.G. and Chesley, O., Chianti: A Tool for Change Impact Analysis of Java Programs. in Object-Oriented Programming Languages and Applications, (Vancouver, British Columbia, Canada, 2004), ACM Press, 432-448.

[3] Ren, X., Ryder, B.G., Stoerzer, M. and Tip, F., Chianti: A Change Impact Analysis Tool for Java Programs. in International Conference on Software Engineering, (St. Louis, Missouri, USA, 2005), ACM Press, 664-665.

[4] Rothermel, G. and Harrold, M.J. A safe, efficient regression testing selection technique. ACM Transactions on Software Engineering and Methodology, 6 (2). 1997. 173-210.

[5] Orso, A., Apiwattanapong, T., Law, J., Rothermel, G. and Harrold, M.J., An empirical comparison of dynamic impact analysis algorithms. in International Conference on Software Engineering, (Edinburgh, Scotland, 2004), ACM Press, 491-500.

[6] Bennett, K.H. and Rajlich, V.T., Software Maintenance and Evolution: A Roadmap. in Future of Software Engineering Track - International Conference on Software Engineering, (Limerick, Ireland, 2000), ACM Press, 75-87.

[7] Staudenmayer, N.A. Managing Multiple Interdependencies in Large Scale Software Development Projects Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, USA, 1997.

[8] Grinter, R.E. Recomposition: Coordinating a Web of Software Dependencies. Journal of Computer Supported Cooperative Work, 12 (3). 1998. 297-327.

[9] Grinter, R. Supporting Articulation Work Using Configuration Management Systems. Computer Supported Cooperative Work, 5 (4). 447-465.

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

[11] McCracken, G. The Long Interview. SAGE Publications, Thousand Oaks, CA, 1988.

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

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

[14] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[15] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. Object Oriented Modeling and Design. Addison-Wesley, Upper Saddle River, NJ, 1991.

[16] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. Pattern-Oriented Software Architecture: A System of Patterns. Wiley, UK, 1996.

[17] de Souza, C.R.B., Redmiles, D., Cheng, L.-T., Millen, D. and Patterson, J., Sometimes You Need to See Through Walls - A Field Study of Application Programming Interfaces. in Conference on Computer-Supported Cooperative Work, (2004), ACM Press, 63-71.

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

[19] Coulon, A. Ethnomethodology. Sage Publications, Thousand Oaks, CA, 1995.

[20] de Souza, C.R.B. On the Relationship between Software Dependencies and Coordination: Field Studies and Tool Support Department of Informatics, Donald Bren School of Information and Computer Sciences, University of California, Irvine, Irvine, CA, 2005, 186.

[21] Perry, D.E., and, H.P.S. and Votta, L.G. Parallel Changes in Large-Scale Software Development: An Observational Case Study. ACM Transactions on Software Engineering and Methodology, 10 (3). 2001. 308-337.

[22] Sarma, A., Noroozi, Z. and van der Hoek, A., Palantír: Raising Awareness among Configuration Management Workspaces. in Twenty-fifth International Conference on Software Engineering, (Portland, Oregon, 2003), 444-453.

[23] Sarma, A., Bortis, G. and Hoek, A.v.d. Towards supporting awareness of indirect conflicts across software configuration management workspaces Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, Atlanta, GA, USA, 2007.

[24] Murphy, G., Notkin, D. and Sullivan, K., Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. in Third ACM SIGSOFT Symposium on the Foundations of Software Engineering, (New York, NY, USA, 1995), ACM Press, 18-28.

[25] Dourish, P. Implications for Design. In ACM Conference on Human Factors in Computing Systems, ACM Press, Montreal, Canada, 2006.