Traceability: Taming uncontrolled change in software development

Krzysztof Kowalczykiewicz, Dawid Weiss
Poznań University of Technology, Project OPHELIA[*]

You did it! You messed with my code!
- FRANK, THE PROGRAMMING GURU

Abstract

Current trends in software engineering focus on processes and methodology of human-controlled change management. In this paper we show that there is a strong need for automated, integrated software development environments, where change propagation would be facilitated not only by humans (even if process-driven), but also by software modules specifically dedicated to this task and spanning over all project artifacts. We prove that such integrated approach has a great impact on software development and we introduce a project, which attempts to fulfill the assumptions presented in this paper.

1. Overview of this paper

This paper deals with software-facilitated change propagation, notification about changes in software development and integration of software tools into a unified development platform. This paper is organized as follows: section two introduces the problem of controlling changes - how to estimate the costs, complexity and impact of a newly introduced change. Section three presents existing solutions: software engineering processes devoted to change management practices and tools with support for tracing project elements. In section four we present our vision of what *traceability* in software development should look like. Section five is a summary of advantages of having such a

[*] Ophelia is a project sponsored by the 5th Framework Program of European Union and jointly developed by a consortium of academic and industry partners. http://www.opheliadev.org

system. Section six lists open problems and issues. Eventually, we summarize the key points of this paper in section seven.

2. The cost of change

Both large and small software projects undergo changes. The cause of change may be either external to the enterprise (i.e. unstable customer requirements, economical constraints), or internal (i.e. detected bugs, replacement of development tools, migration of developers). Regardless of its source, a change introduced to one element of a project usually affects its other parts. If this process is uncontrolled, changes may destabilize the project, leading to problems in keeping up with the schedule, or budget. Uncontrolled changes can also have undesired effects in frustration of developers unable to grasp what is being changed and how it affects other parts of the system.

The larger and more complex a project is, the harder change management gets. At some point, the increasing number of people involved in the project and its growing size create a border line between project management and developers. This is illustrated in figure 1 - individual project members have only small bricks of the project under control. Even as a group, they have the knowledge of a larger part, but not the entire project.

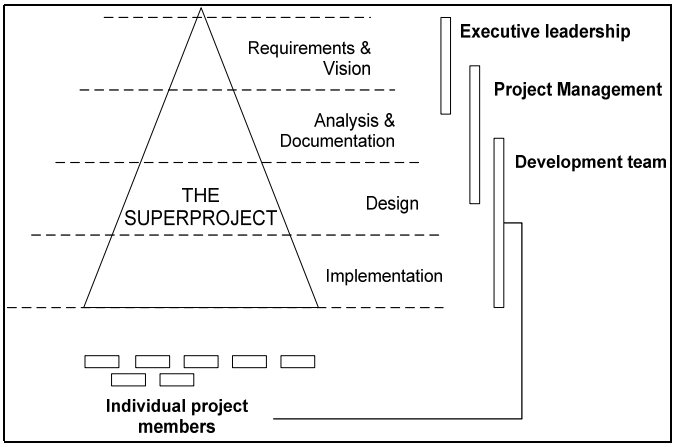

Figure 1. Perception of project participants is usually limited to fragments of the overall project scope.

The decisions taken at management level (which usually involve change acceptance) do not take into account the actual impact on the work to be done by developers, causing underestimation of the effort needed to apply the change to the project, budget-wise or time-wise.

The developers, on the other hand, while usually specialized in their part of the project, are unable to see the overall dependencies of the change they introduce. Although there are development and management processes, which attempt to reduce this gap (see next chapter), they are mostly “paperwork-oriented”, often bringing tedious routine and

mediocre effects ("US Dept. of Defense spends about 4% of its IT costs on traceability, getting inadequate value from that money" [6]). No wonder software projects tend to fail from lack of effective change management more often than from any other reason [1].

3. Existing solutions

Most solutions dealing with change management are based on software engineering processes [2][3].

Traditional change management methods attempt to formalize all activities concerning introduction of changes to the project. Each change must be requested, numbered, accepted, finally implemented and tested to prove it reached its goals. All those activities may be facilitated by specialized document flow tools, yet they still require a human to process the information in each phase. Formal methods are therefore expensive and tedious for their users, but if everything is done according to the rules set up by the organization's change management committee, every change can be tracked and the project should remain consistent throughout its life.

Kent Beck's book "Embracing change" [3] revolutionized thinking of change management in software engineering by breaking the paradigm of formalisms. He introduced the first of the so-called *agile* processes, called simply *eXtreme Programming*. XP methodology attempts to "embrace" change by making sure the separation of concerns in figure 1 does not take place. Rotation of developers, code sharing, pair programming, self-explanatory code - all these to make sure every developer knows just as much about the system as everybody else. Frequent integration and extensive unit tests strategy provide means for ensuring the project is always in a stable state and as bug-free, as possible. However, also XP has some drawbacks. First of all, self-explanatory code may be an excellent practice, but it will not replace the clarity of diagrams and documentation sometimes, especially for a new developer joining the team. Even with the best organized code, a large project is difficult to handle.

As stated in [1], change activities and processes should be supported by software tools. There are a couple of tools with support for tracing relationships among their elements. Most notably, Rational has an integrated environment, which smoothly brings together programs for requirements engineering, design, change management and code repository [2]. Yet, none of the existing tools has support for *all* project elements (ref. to fig. 2), thus forcing users to either use some other, external tools, or go with the paperwork. Also, integrated software packages, like that from Rational, are expensive and binding to a particular vendor, which may not be a good strategy. What we really need, then, is a *cross-tool* integrated environment, which would serve an important purpose: gathering information about associations among project artifacts.

4. The concept of traceability

What seems to be missing in both formal and XP techniques of dealing with change is some repository of relationships among all project elements.

For instance, a piece of code should be linked to its documentation, so that the developer knows that if he changes anything in the code, somebody will be notified to look in the documentation to bring it up to date (even if it's the same person, human memory is a very weak device). A manager may wish to see how many modules or classes a given change will affect, how many resources he must assign to it and, in the end, how much money he will have to request from the client to introduce such change. With the help of associations, he may easily track all project elements originating in the requirement being altered, count them, consult the project members working on that code and finally give a good estimate of the cost of change (refer to figure 2).

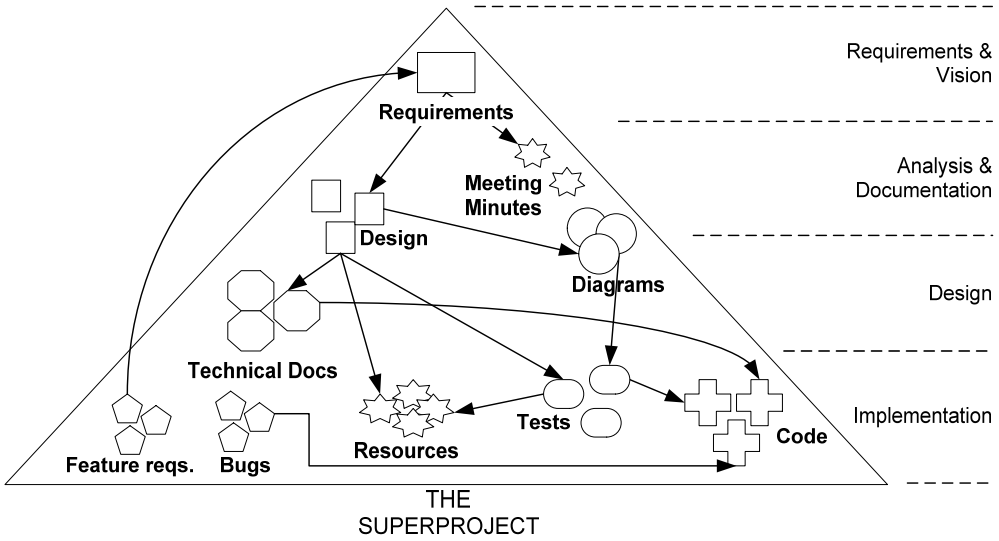

Figure 2. "The superproject" from figure 1, this time a network of relations among its elements is shown. This is solely a presentation of the concept, not all possible associations are depicted.

As mentioned in chapter 2, currently available software tools have very limited support for tracing relations, or cover only parts of the graph in figure 2. Most papers issued on the subject cover the concept tracing associations within requirements and their impact on other project elements. Our concept of *traceability* is about tracing relations among *all* elements, so that associations can be tracked among any given two objects, at any time and are always consistent. This assumption is of course difficult to bring into effect because of the following:

- Tools from different vendors are usually closed applications, with no possibility of integration with other tools.

- New elements of the project and at least some associations among these elements would have to be automatically added to the traceability graph; otherwise it is impossible to keep it consistent and usable.
- Some physical repository of relations is needed; this repository must somehow hold references to objects in proprietary tools' databases.
- Some tools are needed to utilize the relations in the repository (visualize the dependency graph, send notifications about change etc).

While the constraints above render any currently existing software tool or platform unusable to the traceability concept, there is a chance that the situation may change in the future because of a new project developed under the auspices of the European Union, whose objective is to provide a common platform for various software development tools.

4.1 OPHELIA: Tool integration

Ophelia[†] project is an initiative sponsored by the 5th Framework Program of European Union and targeted at specification and development of a tools integration platform especially useful for large, distributed software projects. Many companies nowadays spread all over the world, with specialized divisions or even subsidiary companies producing software components assembled elsewhere. Coordination and management of such projects must be difficult.

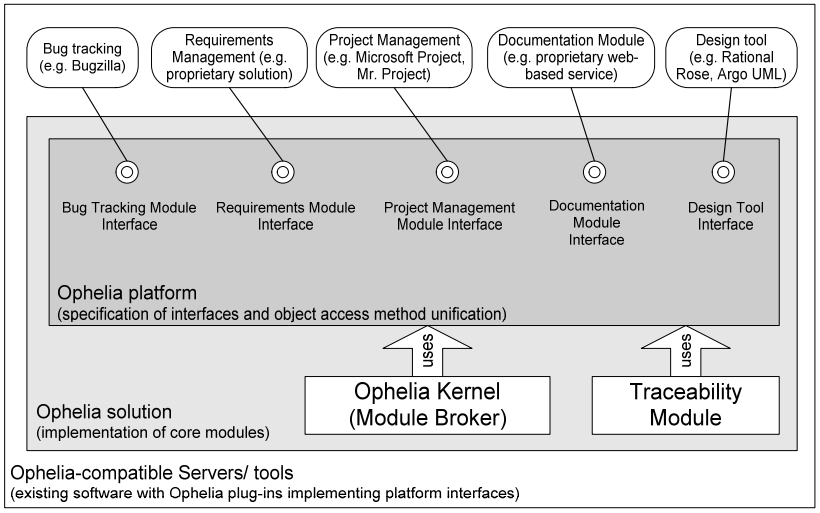

Figure 3. Layers of a traceability-enabled software development environment. The core layer (Ophelia platform) specifies a set of abstract CORBA interfaces. The surrounding layer is a specific instance of the Ophelia environment, with objects/ module broker, security and traceability module. The most outer layer is the layer of software tools used in the development process.

[†] OPHELIA (*Open Platform and metHodologies for devELopment tools IntegrAtion*)

The goal of Ophelia is to propose a definition of a set of CORBA interfaces for various *types* of tools used during project's development, starting at requirements elicitation, ending at documentation and test repositories (see the Ophelia Platform box in figure 3). These interfaces specify abstract functionality of a certain type of tool, but they are not bundled with any particular implementation. Ophelia must have extensive support from tools vendors - they must include an implementation of Ophelia interfaces in their products to allow their coexistence in the integrated platform. The question arises, why would product vendors spend money in implementing such interfaces? The answer to this question being: because Ophelia provides additional services, which otherwise would not be possible: traceability and data integration. The users demanding such functionality should force commercial vendors to include Ophelia plug-ins to their products.

To avoid the egg-chicken paradox some software tools must implement Ophelia interfaces before it is presented to the users to convince them it is a valuable environment. Here is where European Union's project comes into play - as part of the 5th framework project, the Ophelia consortium will develop a fully functional instance of Ophelia called Orpheus, integrating mostly open-source tools, but also some commercial ones, for which Ophelia plug-ins could be written.

Because Ophelia was designed with distributed environment in mind, *Modules* composing the platform work in *client-server* mode. It is not possible to connect an instance of, let's say, ArgoUML, directly to Ophelia. This tool may be connected to a Design Module, which provides all diagrams available in the project, locking of resources (so that users do not work on the same file at the same time), etc. Modules may be provided by tool vendors, or may reuse implementations distributed with Orpheus. Module interfaces are in most cases simple, so that their implementation is not followed with large investment overhead.

4.2 Traceability in Ophelia

All Ophelia interfaces share common definition of an Object. Components of a particular instance of Ophelia may therefore gain information about objects stored anywhere within that instance, regardless of what type of object it is, or which module stores it. *Traceability Module* makes use of this feature to store relations among objects in the system (figure 3 illustrates this concept).

There are two ways of adding new relations to the traceability module:

- **Explicitly defined relations**. It is always best, when a human being tells the system about an explicit relation between two elements. It can also be done with little effort; if done incrementally. When new elements are added (documents, classes etc.) to the project, one should set up relations to other objects already present in the system.
- **Implicitly defined, external relations**. Sometimes it would be more elegant if software created relations instead of users. One may envision "synchronization tools", which we call integrators, utilizing common platform's interfaces and synchronizing data between code and documentation modules in an automatic

fashion (i.e. automatically updating JavaDocs in the technical documentation). The newly created elements could be automatically related to their dependent objects. A basic version of such functionality is planned for the Ophelia instance called Orpheus.

- **Implicitly defined, in-module relations.** Common platform modules would usually include some sort of hierarchy of their objects. Code is kept in modules or packages, documentation in folders, requirements in some dedicated groups perhaps - these structures could also be automatically reflected onto relations present in traceability module, so that users can link other objects, or sign up for notifications on to the entire hierarchy.

The process of collecting relations is continuous with the development of the project; with new elements being added in each phase (or iteration) of the lifecycle, new relations must be added to ensure consistency in the relations database. Yet, this process is much less time-consuming than in the case of creating a formal list of dependencies in the documentation as it should be the case with non-integrated change management processes. The relations in Ophelia may be browsed online, printed, or automatically included in the generated documentation.

4.3 Automatic notifications about changes in Ophelia

So far we have been talking only about creating relations and tracing the graphs of interdependent objects. Another potential use of the traceability relations stored in the integrated platform is in automated change propagation and notifications.

Developers know from experience that even a very small change to the code causing a bug is a beast with a tendency of sneaking completely unnoticed until it is too late to fix it. Especially in geographically distributed projects, this is a common source of frustration for programmers, that they are not in control of what has been changed over night (on the other hemisphere)...

Notification mechanism brings an ideal solution to this problem. Users may “sign up” to receive notifications about changes of some object in the integrated environment. Because of uniform definition of core platform interfaces, and the distributed nature of Ophelia, all objects can be accessed, located and used by the notification mechanism as if they were local (even though they come from various tools, perhaps originating in different physical locations). So, the documentation-writer in Brazil may sign up to receive notifications about changes in any of the source files of a module. He will be automatically notified if the developer in Poland fixes a bug and thus changes the source code anywhere.

But notifications are only part of the power that an integrated environment like Ophelia brings. If notification is coupled with traceability, then information about a change may be propagated along object relations. This gives a unique opportunity of controlling changes on a hierarchy of objects, for example the entire group of requirements, or a test unit.

The main innovation of the above solution is in the fact that the entire project, with all its artifacts, becomes a scope, unlike with existing products, where traceability and change management is usually limited to objects in a certain product only.

4.4 Orpheus project work status

The Ophelia module interfaces have been released in alpha phase and are available via the project's website (*http://www.opheliadev.org*). Implementation of servers and client plug-ins of Orpheus, the Ophelia instance, is in progress. First public alpha release of the platform is planned to happen in Fall, 2002.

5. Summary of key advantages of integrated traceability

A full discussion of the advantages of having an opportunity to trace changes and relationships among elements of a project is beyond the scope of this paper. We would like to summarize the most important factors below.

- Tracing relations among project artifacts is a difficult issue, requiring time and effort if done according to software engineering guidelines. This process could be greatly facilitated using software tools. Even though no options of integrating software utilities from different vendors currently exist, such platforms may emerge in the future (e.g. Ophelia, Eclipse).

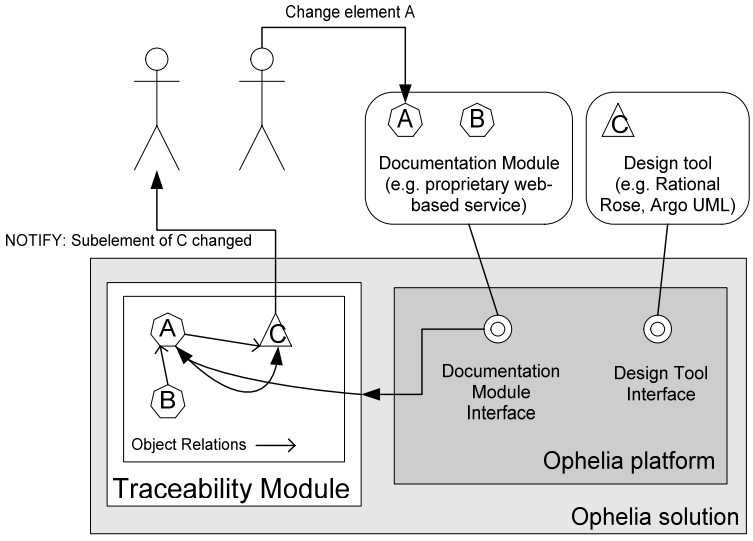

Figure 4. Notifications can be propagated upon related objects.
Users may track changes of the entire logical structure of project elements.

- Integrated traceability creates a new viewpoint on the project: a viewpoint on all of its elements and their relations, regardless of their type, or tool used for their

editing. Project members can easily understand and measure the size of a particular element's dependency graph and its relation to other parts of the project. Traceability becomes a kind of a roadmap to the project.

- Change management is greatly facilitated by combining traceability and messaging - users can be notified about object changes. They can also request to be notified every time any element of a given hierarchy changes (i.e. notify me when any of this module's source files changes). It is easy to imagine adding an auditing mechanism to this scheme, so that users are not notified about a change, but are also required to approve it. A person responsible for updating documentation would have to bring it up to date before approving the notification originating from the source code, for example.
- A platform for tools integration provides many additional features besides traceability; for example, one could imagine utilities for automated data synchronization, project metrics calculated on elements of different types, or only on the semantics of their dependencies (analogous to the OO class coupling metric).

6. Open problems and issues

As with any new idea, there are also numerous problems to be solved. We summarize these problems in this chapter.

- With no support from software tools vendors the entire idea of tool integration will eventually fail (all tools used in the project must be integrated in order for the concept of traceability to work). It is not impossible, however to imagine such support. There are symptoms already - Microsoft .NET platform, integration of tools into Eclipse IDE, or even coupling of compilers and debuggers into visual programming environments - a concept revolutionary 15 years ago and so natural now. On the other hand, tool vendors would like to have their market advantage over other products - the need of interfaces unification is a contradictory axis here and some tradeoff will have to be found.
- Traceability graph may become dense and cluttered with junk relations. Taking an analogy to cartography, one never puts together city maps if a continent overview is needed. A *scale* may be a problem - adding too many detailed relations may obstruct the view of the overall project architecture. Also, too many relations may cause each change to cause an avalanche of cascading notifications.
- Most relations are added explicitly to the system by project members. Some procedures should be available so that rules on when, how and which objects should be added to traceability are clear. Otherwise there is a danger that either too many, or too few relations are created, both of which are situations that should be avoided. In other words, having traceability software will not replace a good software engineering process, but it is going to make it easier for project members to comply with the rules.

7. Summary

In this paper we have presented an overview of an idea of tracing relationships among all of project artifacts, We showed the possible uses of such relationships and an example architecture, which would allow this concept to be used with existing development tools. We also introduced the Ophelia project, which is an implementation of the traceability-enabled architecture.

The problem of facilitating change management and tracking relations in large, often distributed, software projects still remains open. The traceability concept and environments like Ophelia bring a solution of consolidating software development activities, give a new perspective on impact analysis and change management in general.

References

[1] *An overview of Change Management.* [@:] http://www.prosci.com/Change %20management%20overview.htm (05/2002).

[2] *Rational Change Management*. [@:] http://www.rational.com/leadership/initiatives/ucm.jsp (05/2002).

[3] Beck K.: *Extreme Programming Explained: Embrace Change*. Addison Wesley Longman, Reading Mass. 2000.

[4] Hapke M., Jaszkiewicz A., Kominek P.: *Zintegrowane narzędzia harmonogramowania przedsięwzięć programistycznych w warunkach niepewności*. Materiały I Krajowej Konferencji Inżynierii Oprogramowania, Kazimierz Dolny, 1999.

[5] Hapke M., Jaszkiewicz A., Perani S.: *OPHELIA - Open platform and methodologies for development tools integration in a distributed environment*. In Proceedings of 3rd National Conference on Software Engineering, Otwock/Warsaw, pp. 189-198.

[6] Ramesh B., Jarke M.: *Towards reference models for requirements traceability*. [@:] http://citeseer.nj.nec.com/ramesh99towards.html (05/2002).

[7] Arnold R., Bohner S.: *Impact Analysis - Towards a Framework for Comparison*. Proceedings of the International Conference on Software Maintenance, 1993.

[8] MacKinnon L. M.: *The Ophelia Project*. The British HCI Group & Scotland IS Usability Forum, Usability and UML Symposium, Edinburgh , 2002.