

Modeling software evolution by evolving interoperation graphs^{*}

Václav Rajlich

Department of Computer Science, Wayne State University, Detroit, MI 48202, USA
E-mail: rajlich@cs.wayne.edu

Software evolution is the process of software change, most often change in software requirements. This paper presents a theoretical model for the evolution of component-based software, based on evolving interoperation graphs. The model assumes that each change consists of smaller granularity steps of change propagation, each of them being a visit to one specific component. If the component is modified, it may no longer fit with the other components because it may no longer properly interact with them. In that case secondary changes must be made in neighboring components, which may trigger additional changes, etc. The paper contains an example of evolution of a calendar program, represented in UML.

1. Introduction

Software changeability (evolvability) is one of the essential properties of software [Brooks 1987]. All successful software continually changes in response to changing user expectations, changing hardware, changing environment in which software operates, etc. The fact of constant software change and evolution was well documented in [Lehman *et al.* 1998]. Because of constant change, changeability plays a key role in both current and future software technologies.

Because of the importance of change, it is a significant goal for research to make software changes easy, safe, and inexpensive. One of the solutions [Parnas 1972] is to anticipate changes and build the software in such a way that the changes will be localized inside components. When a change is localized within a component, it is easier, safer, and less expensive. However, more recent research indicates that not all changes in the software can be anticipated. In the case study of [Cusumano and Selby 1997] it was reported that approximately 70% of the requirements were predicted in advance and the remaining requirements were discovered during development. Hence the software was already undergoing changes during development, because the requirements were not accurately predicted. We speculate that the changes during software maintenance are even less likely to be accurately predicted. The current software trend is the development of applications of ever increasing complexity and heterogeneity and

^{*} This work was partially supported by a grant from Ford Motor Co. and NSF grant #CCR-9803876.

this will make accurate prediction of changes even harder. It is likely that these applications will be exposed to many unanticipated changes during their lifetime. Therefore better support for unanticipated changes is an important research goal.

In this paper, we are presenting a model for software evolution. The development of a model is seen as the first step towards tools that would support changes in software. A model of software evolution must capture discrete time, because the evolving application changes in discrete time steps. The model must deal with interacting components, because interacting components are a fundamental aspect of complex applications. The interactions and components are of a great variety [Shaw and Garland 1996] and hence the model must be abstract in order to be universal. The model must capture the fact that a change affects both the components and their interactions. An important characteristic of software change propagation is locality, because changes typically affect only a part of application and leave the rest of the system intact. Finally, it is desirable that the model be lucid, without unnecessary clutter, yet complete enough so that it captures essential properties. The model presented in this paper has these properties.

The model assumes that each software change starts and ends with consistent software that can be compiled, executed, or used. Each change consists of smaller granularity steps of change propagation, each of them being a visit to one specific component. If the component is modified, it may no longer fit with the other components because it may no longer properly interact with them. In that case secondary changes must be made in neighboring components, which may trigger additional changes, etc. Hence during change propagation, software is often inconsistent and requires additional changes. The process of change propagation ends only when all inconsistencies are removed.

This model is explained in the paper and illustrated by an example of the evolution of a calendar notebook. Section 2 of the paper contains the definitions of the evolving interoperation graphs that are used to model software evolution. Section 3 contains the example. Section 4 describes related work. Finally, conclusions and future developments are found in section 5.

2. Evolving interoperation graphs

Software consists of components and their interoperations and is formally modeled by interoperation graphs. Let C be a set of *components* of the program. An *interoperation* or *interaction* between two components $b, c \in C$, $b \neq c$ is formally represented as an unordered couple $\{b, c\}$. It intuitively means that the two components interact in some way and if one changes, the other one may be affected too. An *interoperation graph* G is a set of interoperations.

Let G be an interoperation graph, then interoperation $\{b, c\} \in G$ is *marked* if there exists an ordered couple $\langle b, c \rangle$ called *mark*. Intuitively it means that component b has been changed or inspected in the past and component c will be changed or inspected in the future. Marked interaction can be inconsistent, i.e., there could be a conflict

between what one component provides and what the other component requires. If $\langle b, c \rangle \in E$, then c is called *marked* component. *Evolving interoperation graph* (eig) is a set E of interoperations and marks such that $\langle b, c \rangle \in E$ implies $\{b, c\} \in E$. Eig E is *unmarked* if $\{\langle d, b \rangle \mid \langle d, b \rangle \in E\} = \emptyset$ and *marked* if $\{\langle d, b \rangle \mid \langle d, b \rangle \in E\} \neq \emptyset$.

Let E be an eig, then $\text{comp}(E) = \{b \mid \text{there is } \{b, c\} \in E\}$ is the *underlying set of components*. For a component $b \in \text{comp}(E)$, define the following sets:

$$\begin{aligned} G(b) &= \{\{b, c\} \mid \exists c, \{b, c\} \in E\} && \text{(interoperation neighborhood),} \\ \underline{M}(b) &= \{\langle c, b \rangle \mid \exists c, \langle c, b \rangle \in E\} && \text{(incoming marks),} \\ M(b) &= \{\langle b, c \rangle \mid \exists c, \langle b, c \rangle \in E\} && \text{(outgoing marks),} \\ E(b) &= G(b) \cup \underline{M}(b) \cup M(b) && \text{(evolving neighborhood).} \end{aligned}$$

A *visit* to a component is the replacement of that component and its neighborhood by an updated one. This is formally defined in the following way: Let b and b' be a component before and after the visit, E and E' be an eig before and after the visit, and $E(b)$ and $E'(b')$ be the neighborhood of the visited component before and after the visit, respectively. Then a *visit* is a couple of eigs $\langle E, E' \rangle$ such that $E' = (E - E(b)) \cup E'(b')$. Please note that components and interactions outside the neighborhood of the visited component b are not changed by the visit.

A *scenario* is a sequence of evolving dependency graphs E_1, E_2, \dots, E_n which starts and ends with unmarked graphs, i.e., both E_1 and E_n are unmarked. If there are no backtracks, then for each step i , $\langle E_i, E_{i+1} \rangle$ is a visit. If there are backtracks in the scenario, then either $\langle E_i, E_{i+1} \rangle$ is a visit, or for some k , $0 < k < i$, $E_{i+1} = E_k$.

Each scenario employs a specific *strategy* that depends on specific circumstances of the project. Here we will consider finality and strictness as the two fundamental aspects of the strategies. If visits are *final* then we assume that after visiting first component b and then its neighbor component c , component b does not need another immediate revisit, i.e., the change does not bounce back. If it is not final, then there may be a need to revisit b immediately again. This is described in the following notation:

- For a change that is not final, $\underline{M}'(b') = \emptyset$ and $M'(b') \subseteq \{\langle b', c \rangle \mid \{b', c\} \in E'(b')\}$.
- For a final change, $\underline{M}'(b') = \emptyset$ and $M'(b') \subseteq \{\langle b', c \rangle \mid \{b', c\} \in E'(b')\} - \{\langle b', d \rangle \mid \langle d, b \rangle \in \underline{M}(b)\}$.

If there is only a static analysis without any knowledge of semantics of interoperations, we will distinguish only two possibilities: Either the change propagates through all interoperations, or it does not propagate at all. In the first case, for non-final change, $M'(b') = \{\langle b', c \rangle \mid \{b', c\} \in E'(b')\}$, and for final change, $M'(b') = \{\langle b', c \rangle \mid \{b', c\} \in E'(b')\} - \{\langle b', d \rangle \mid \langle d, b \rangle \in \underline{M}(b)\}$. In the second case, $M'(b') = \emptyset$.

Another characteristic that distinguishes strategies is their *strictness*. Strict strategies start with the first component, and after that they visit only marked components. This contrasts with random or lenient strategies, where any component can be visited any time.

A special case is introduction of a new component b' into the eig. In that case, $E' = E \cup E'(b')$, where $E'(b') = M'(b) \cup G'(b')$ and $M'(b) = \{\langle b, c \rangle \mid \{b, c\} \in G'(b)\}$. (All interactions of the new component are marked.)

Deletion of a component from the model has to be handled in two steps. In the first step, all neighbors of the component are marked, i.e., $E' = (E - E(b)) \cup E'(b)$, where $G(b) = G'(b)$, $\underline{M}'(b) = \emptyset$ and $M'(b) = \{\langle b, c \rangle \mid \{b, c\} \in G(b)\}$. In the second step, all marked components are changed so that they no longer make any reference to the component being removed, following final policies for all of them.

3. Example

In this section, we show an example of software evolution of a calendar program. For simplicity, the scenarios of the example use final and strict strategy.

Class diagrams of Unified Modeling Language (UML) [Booch *et al.* 1998] are used to represent the program. The classes of UML are represented by components of eig, and associations and dependencies of UML are represented by interactions of eig. If there is an association or dependency between classes b and c of UML, then there is also interoperation between b and c of the corresponding eig. For n -ary association between classes c_1, c_2, \dots, c_n there is an interoperation for every couple of classes involved in the association, i.e., $\{c_1, c_2\}, \dots, \{c_1, c_n\}, \dots, \{c_{n-1}, c_n\}$. For an association class c_A , there is an interoperation between c_A and all classes in the association, i.e., $\{c_1, c_A\}, \{c_2, c_A\}, \dots, \{c_n, c_A\}$. Each class diagram in UML can be converted into an eig by these rules.

The example of this section deals with the evolution of a program that implements an appointment book. It consists of four scenarios of change propagation, each with several visits. The first scenario introduces the basic program and the following scenarios add new features to it.

3.1. Simple appointment notebook

This scenario starts from scratch and ends with a simple appointment notebook that maintains a list of appointments for one week. The user can add, delete, and search appointments. When adding a new appointment, the program will check whether the new appointment conflicts with any already existing appointments.

During this scenario, several steps of change propagation are employed and they all introduce new classes. The scenario is in fact the same scenario as the one described in [Rajlich 1994], where the newly introduced classes were called deferred classes. Their characteristic feature is that they contain only a class interface, and the implementation is missing.

The first class introduced is the class `user` and its neighborhood. Class `user` has operations `add`, `delete`, and `search` and these operations manipulate a data structure in a neighboring class `eventList`. The resulting eig $E = \{\{user, eventList\}$,

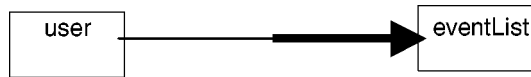


Figure 1. Introduction of class user.

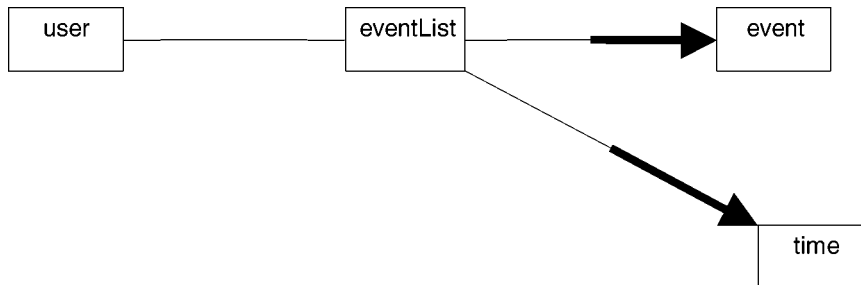


Figure 2. Visit of class eventList.

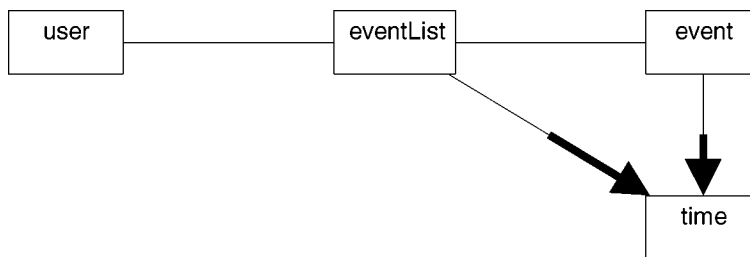


Figure 3. Visit of class event.

$\langle \text{user}, \text{eventList} \rangle$ is given in figure 1. The mark $\langle \text{user}, \text{eventList} \rangle$ is denoted by arrow. It means that `eventList` has to be visited in the future and in practice it means that class `eventList` is deferred and contains only a class interface.

In the next step, class `eventList` is changed (i.e., fully implemented) and new classes `event` and `time` are added as neighbors. Class `event` contains data of one appointment. Formally, the new eig is $E = \{\{\text{user}, \text{eventList}\}, \{\text{eventList}, \text{event}\}, \{\text{eventList}, \text{time}\}, \langle \text{eventList}, \text{event} \rangle, \langle \text{eventList}, \text{time} \rangle\}$, see the corresponding graph in figure 2.

In the next step, class `event` is fully implemented. The change introduces interoperation of `event` and `time` because each event has the beginning and ending time. The resulting eig is given in figure 3.

Finally, class `time` is visited and completed and the resulting eig is given in figure 4. This eig is unmarked and therefore the scenario is complete. The corresponding UML class diagram is given in figure 5.

When the scenario is finished, the code can be compiled and executed and represents the solution of the problem on the current level of detail. The next three scenarios will add new features to this model.

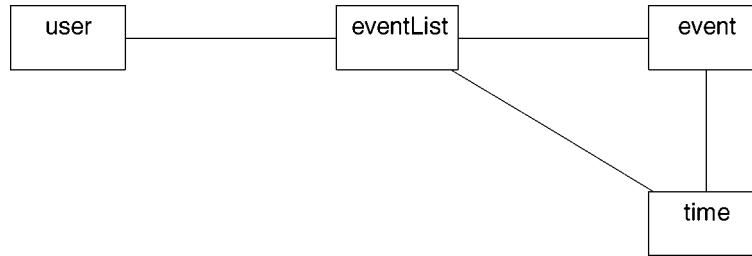
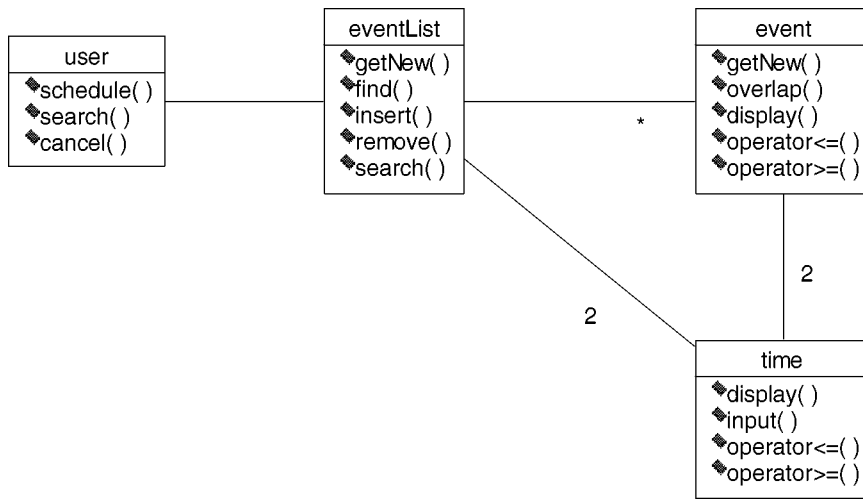
Figure 4. Visit of class `time`.

Figure 5. UML class diagram after first scenario.

3.2. Adding nested events

The previous scenario created a simplified version of the calendar program. In the second scenario, we add a new feature to the program: nested events that contain other events. An example is a business trip that can have appointments within it.

In the first step of this scenario, we observed that classes `event` and `eventList` deal with the same objects, one being a composition (list of events) and the other one an individual object (single event). The Composite pattern [Gamma *et al.* 1995] is the appropriate solution in this situation. In order to introduce the Composite pattern, we created a new class `eventAbstr` and added it to the program, see the eig in figure 6. Class `eventAbstr` was specified to be the base class to both `event` and `eventList` classes. Some operations that were formerly members of either `eventList` or `event` were added to `eventAbstr`. Also the interoperation with `time` was added to `eventAbstr`. Since `eventAbstr` is a new class, all interoperations with all neighboring classes are marked, see figure 6.

After the addition of `eventAbstr`, there are three marked classes in the eig. The first visited class is `eventList`. The visit deletes all interoperations of `eventList`

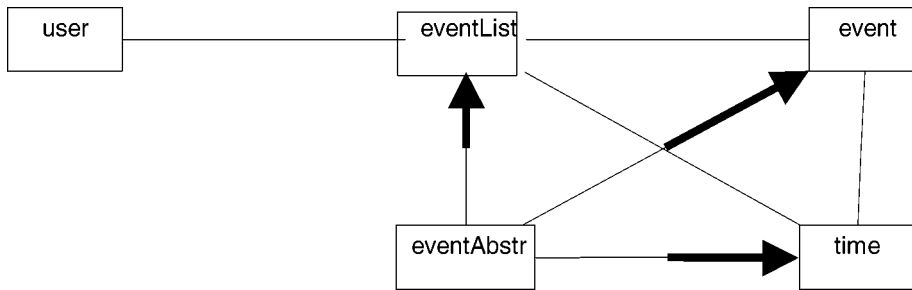


Figure 6. Introduction of class eventAbstr.

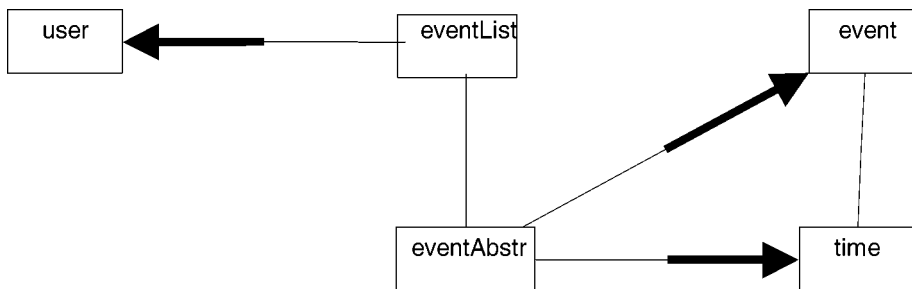


Figure 7. Visit of class eventList.

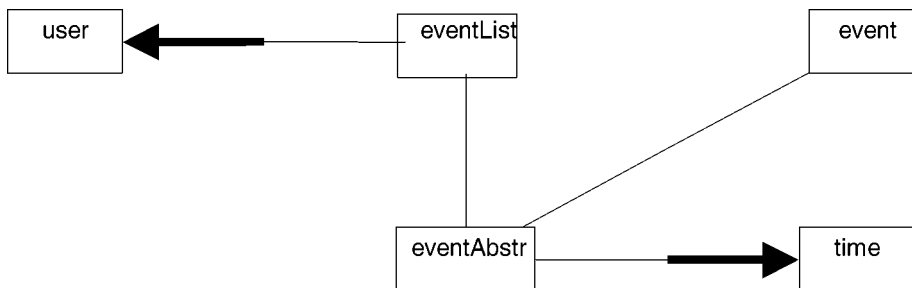


Figure 8. Visit of class event.

with `event` and `time` that are now inherited from `eventAbstr`. Also operations that are now inherited from `eventAbstr` are deleted from `eventList`. The interoperation between `eventList` and `user` is marked because of the changes in `eventList`, see the resulting eig in figure 7.

The next class visited is class `event`. Interoperation with `time` is removed because it is now inherited from `eventAbstr`, and also some operations now inherited from `eventAbstr` are removed from `event`. The resulting eig is given in figure 8.

Finally, classes `user` and `time` are visited. Class `time` does not require any change and class `user` needs changes to operations that deal with nested events. The resulting UML class diagram is given in figure 9.

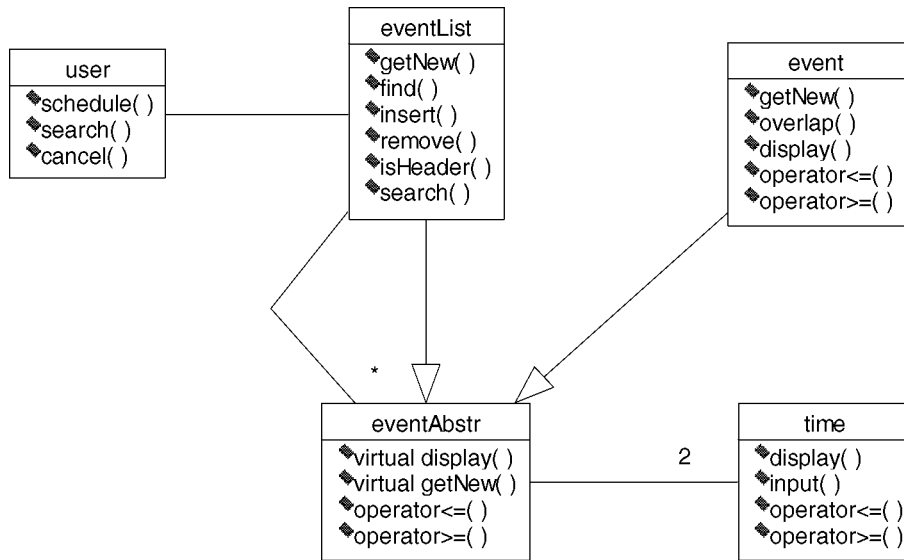


Figure 9. UML class diagram after second scenario.

3.3. Adding multiple users

In some situations, multiple users access the appointment book. For example, there may be a secretary who enters appointments into the block of time set aside for office hours. Multiple users and their authorizations are the new features added in this scenario. The scenario starts with adding a new class `userMain` that supports the master user. It maintains a list of other users and their authorizations, see the eig in figure 10.

The next step is the revision of class `user`. Class `user` now deals with authorization, therefore a new deferred class `authorization` is introduced as one of `user`'s neighbors. The change also propagates to `eventList`. The resulting eig is given in figure 11.

In the next step, class `eventList` is visited. It does not require any changes to its own code, but propagates the change to `eventAbstr`. Intuitively it means that although `eventList` does not change, it expects new services from interoperating class `eventAbstr`, in this case handling the authorization codes for individual events. After the visit, the new eig is given in figure 12.

Class `eventAbstr` is the next class selected for visit. Insertion of events into the appointment book now requires authorization, therefore a new interoperation between `eventAbstr` and `authorization` is established and neighbors of `eventAbstr` are marked, see the eig in figure 13.

Finally, classes `event` and `time` do not require change, and class `authorization` is changed to fit with the rest of the program. After that, the eig is consistent again. The UML class diagram of the final program is given in figure 14.

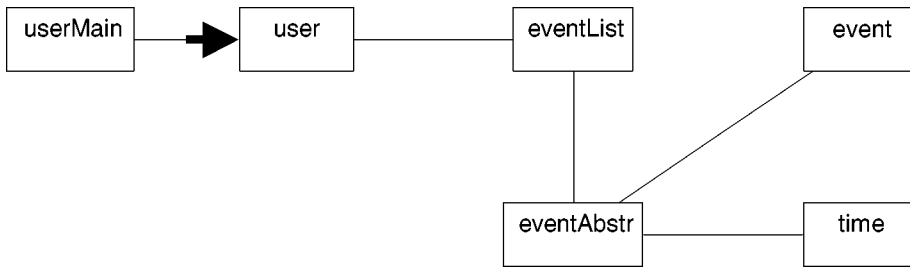


Figure 10. Introduction of class userMain.

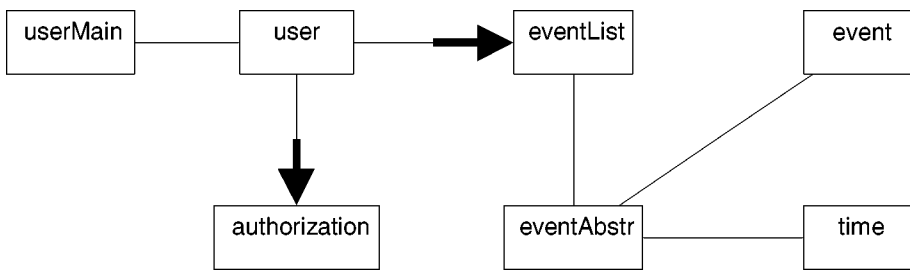


Figure 11. Visit of class user.

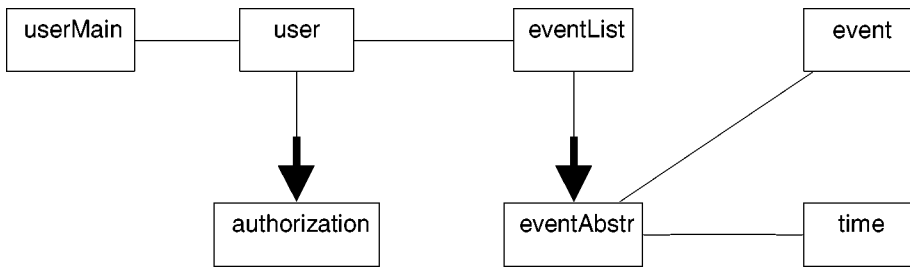


Figure 12. Visit of class eventList.

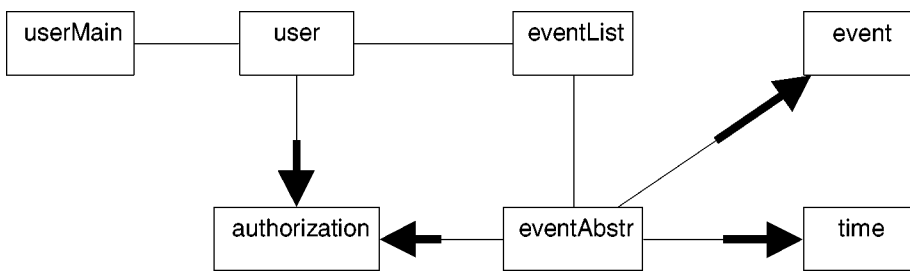


Figure 13. Visit of class eventAbstr.

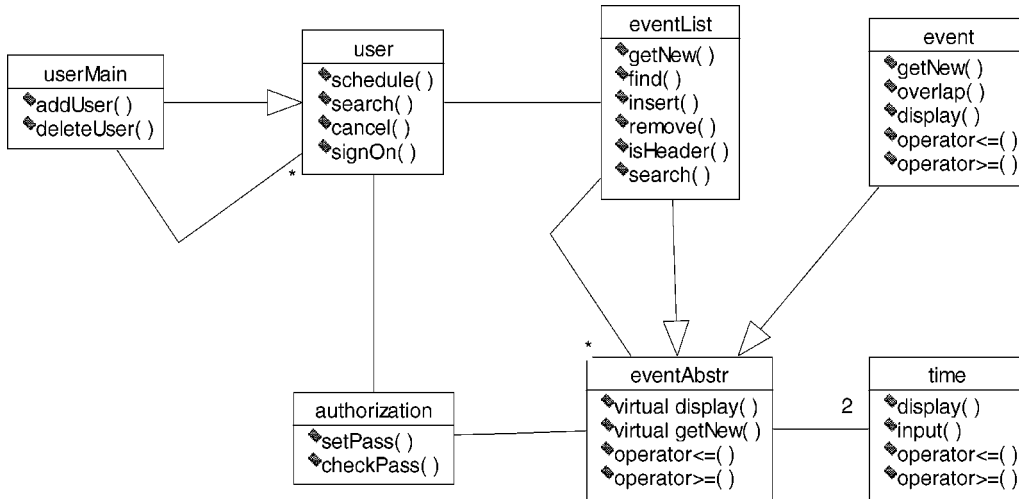


Figure 14. UML class diagram after third scenario.

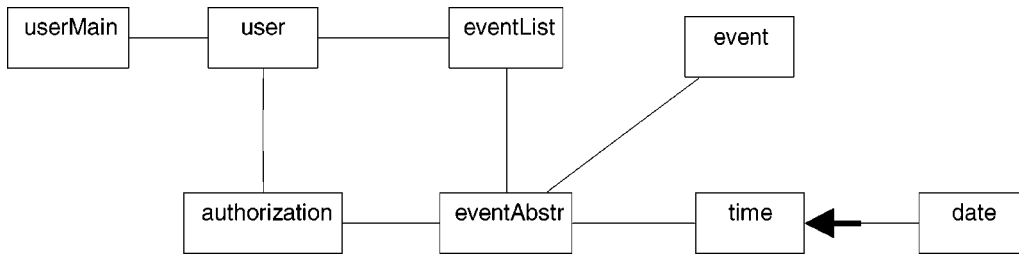


Figure 15. Introduction of class date.

3.4. Extending the scope

In this scenario, we are going to extend the scope of the calendar beyond one week. After the scenario, the calendar will be able to deal with arbitrary dates from year 1999 until year 9999. The calendar will treat weekends differently than the rest of the days and warn the user that a particular event involves a weekend. A new class `date` will be used for that. It will check the correctness of a date entered by users and it will also check whether the date is a weekend. The new eig of the program is given in figure 15.

Class `time` has to be changed because it now makes a reference to `date`. The change propagates to `eventAbstr`, see figure 16.

Class `eventAbstr` itself does not change, but the changed semantics involves handling of dates and propagates through it to the neighboring classes, see figure 17. It will not affect classes `event` or `authorization`, but through `eventList` it will propagate to `user` who will now get a warning when attempting to schedule events on weekends or holidays, see figure 18. After class `user` has been updated, the scenario is complete and the final UML class diagram is given in figure 19.

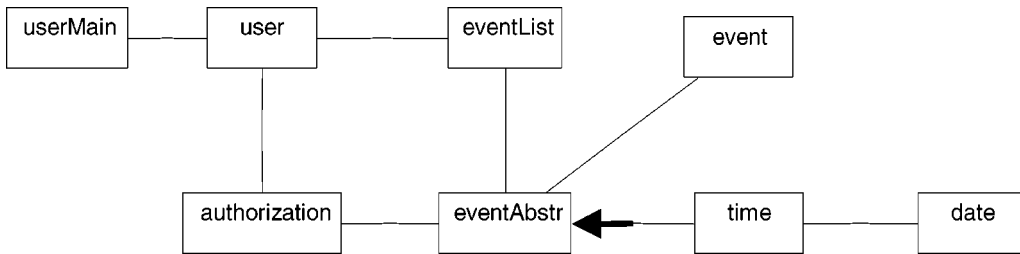


Figure 16. Visit of class time.

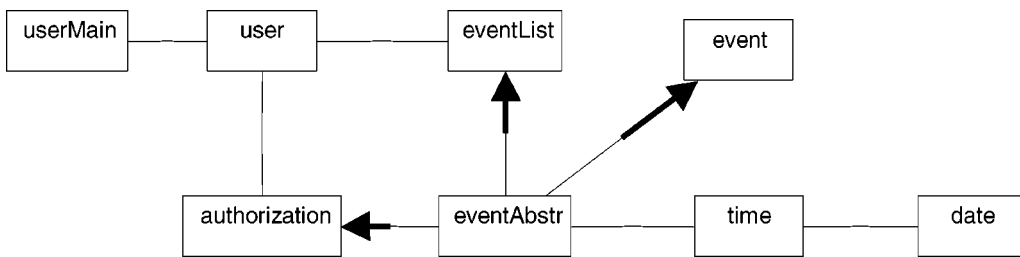


Figure 17. Visit of class eventAbstr.

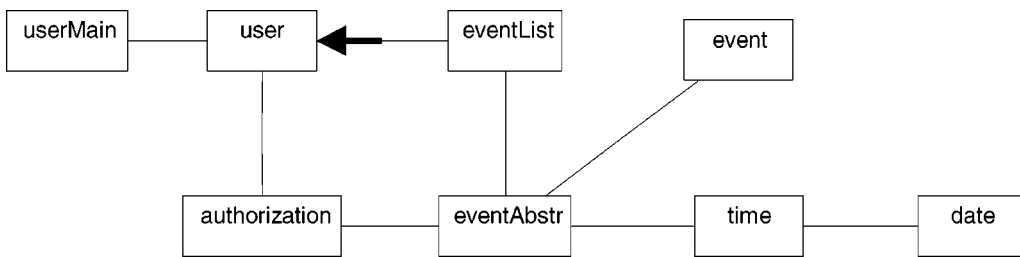


Figure 18. Visit of classes authorization, event, eventList.

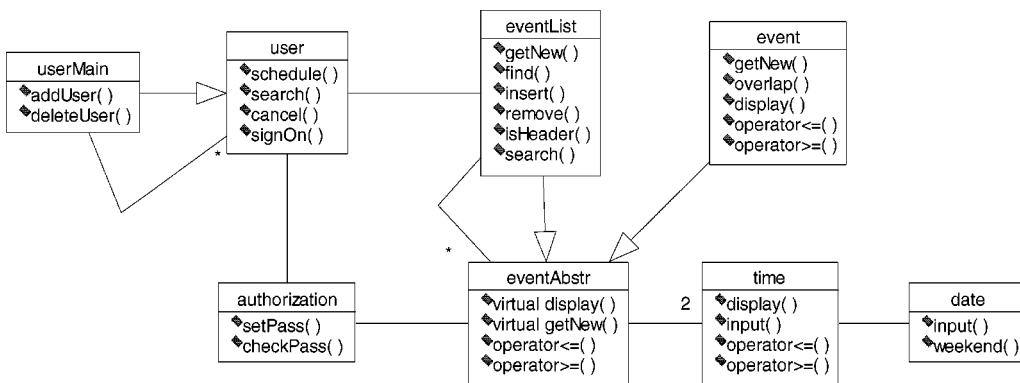


Figure 19. Final UML class diagram.

4. Related work

A well-balanced selection of research on software evolution and change impact analysis appeared in [Bohner and Arnold 1996]. Several papers in the selection deal with program analysis, where the actual code is analyzed and important properties are extracted, most commonly control flow and data flow. The analysis tools create a graph representing these relationships. Since an unmarked eig is based on a subset of information needed from these graphs, the same analysis tools can be used to extract an eig from the code.

The work presented in this paper is related to the work on software architectures. According to [Shaw and Garland 1996], the architectures involve description of components from which the systems are built, their interactions, patterns that guide their composition, and constraints on these patterns. Based on that definition, an unmarked eig is a part of a software architecture. A marked eig then contains additional information about the current state of change propagation, represented by the marks. Browsers are the software tools that extract architectural information from legacy code [Chen *et al.* 1990; Rajlich *et al.* 1990] and store it in a database. They extract the information that can be used in order to construct the eig of a program.

Once the architectural information has been extracted, it can be used in various software algorithms and scenarios. One of the best known examples of such use is smart recompilation [Tichy 1994] that uses component dependencies to suggest recompilation of the components that depend on a previously changed component. Smart recompilation provides a highly specialized change propagation algorithm for a bottom-up change propagation, i.e., change propagation from providing component *b* to using component *c*. If providing component *b* changes, then requiring component *c* must be visited and possibly changed. Compared to that, our change propagation model is more universal in at least two ways: It also deals with top-down change propagation, like in scenario 1 and scenario 3 of the example in section 3, where the change starts at the top and propagates from top to bottom. The other situation is the fact that the changes can propagate through a component without changing it, like in scenario 3 where the change propagates through component *eventList* without changing it but changing neighboring components. That situation would cause smart recompilation to stop at `eventList`.

Another related work deals with the change impact analysis [Li and Offutt 1996; Queille *et al.* 1995; Yau *et al.* 1978]. The purpose of the change analysis is to discover which components are going to be affected by the change, before the actual change is made. Hence change impact analysis is a part of change planning, rather than actual change implementation. Queille *et al.* [1995] describe a tool that takes into account the probability of propagation from one component into another one. Li and Offutt [1996] list various changes to a C++ class and the possible impacts of such changes.

There are several papers where the scenarios of change propagation were investigated [Luqi 1990; Rajlich 1997a,b]. Compared to this earlier work, the model

presented in this paper is more general and aimed at the fundamentals of the strategies of propagation.

The formalism of the paper is based on graph rewriting as presented in [Rajlich 1977] and later in [Metayer 1998] and several other publications.

The scenarios described here are equivalent to iterations of iterative software life cycle, as discussed in [Booch *et al.* 1998] and many other publications. These discussions emphasize the advantages of iterative life cycle where software is developed one scenario at a time, but they do not study detailed principles of change propagation on which iterative life cycle must be based. They assume that the change will be done intuitively by an experienced programmer.

Evolvability of software architectures was studied in [Rajlich and Silva 1996; Rajlich and Ragnathan 1998].

The work on version control and configuration management [Tichy 1994] tacitly assumes that the program is consistent and it in fact addresses the issue of how to manage the results of completed changes. It does not deal with the finer granularity of change propagation steps and inconsistencies in software during these steps.

There can be additional scenarios of change propagation, see for example [Rajlich 1997b] where the scenarios were classified based on specifics of interoperations.

5. Conclusions and future work

In this paper, the formalism of the eig was used to study properties of scenarios of software evolution. Future work involves developing a software tool that supports the scenarios. The tool will extract the eig from the program, and manipulate the marks in order to support the evolution. In this way, it will support the programmer in the task of software change.

Another topic for future work is the evolvability of software architectures. Evolvable architectures support scenarios with short sequences of change propagation. An example is the orthogonal architecture of [Rajlich and Silva 1996].

References

- Bohner, S.A. and R.S. Arnold (1996), *Software Change Impact Analysis*, IEEE Computer Society Press, Los Alamitos, CA.
- Booch, G., J. Rumbaugh, and I. Jacobson (1998), *The Unified Modeling Language User Guide*, Addison-Wesley, Reading, MA.
- Brooks, F. (1987), "No Silver Bullet," *IEEE Computer* 20, 4, 10–19.
- Chen, Y.F., M.Y. Nishimoto, and C.V. Ramamoorthy (1990), "The C Information Abstractor System," *IEEE Transactions on Software Engineering* 16, 325–334.
- Cusumano, M.A. and R.W. Selby (1997) "How Microsoft Builds Software," *Communications of ACM* 40, 6, 53–61.
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides (1995), *Design Patterns*, Addison-Wesley, Reading, MA.

- Lehman, M.M., D.E. Perry, and J.F. Ramil (1998), "Implication of Evolution Metrics on Software Maintenance," In *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 208–217.
- Le Metayer, D. (1998), "Describing Software Architecture Styles Using Graph Grammars," *IEEE Transactions on Software Engineering* 24, 3, 521–533.
- Li, L. and A.J. Offutt (1996), "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software," In *Proc. International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 171–184.
- Luqi (1990), "A Graph Model for Software Evolution," *IEEE Trans. on Software Engineering* 16, 8, 917–927.
- Queille, P.P., J.F. Vodroit, N. Wilde, and M. Munro (1995), "The Impact Analysis Task in Software Maintenance: A Case Study," In *Proceedings of International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 235–252.
- Parnas, D.L. (1972), "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM* 15, 12, 1053–1058.
- Rajlich, V. (1994), "Decomposition/Generalization Methodology for Object Oriented Programming," *Journal of Systems and Software* 24, 181–186.
- Rajlich, V. (1977), "Theory of Data Structures by Relational and Graph Grammars," In *Automata, Languages, and Programming*, Lecture Notes in Computer Science, Vol. 52, Springer-Verlag, Berlin, pp. 391–511.
- Rajlich, V. (1997a), "MSE: A Methodology for Software Evolution," *Journal of Software Maintenance* 9, 103–125.
- Rajlich, V. (1997b), "A Model of Change Propagation based on Graph Rewriting," In *Proceedings of IEEE International Conference on Software Maintenance*, pp. 84–91.
- Rajlich, V., N. Damaskinos, P. Linos, and W. Khorshid (1990), "VIFOR: A Tool for Software Maintenance," *Software – Practice and Experience* 20, 1, 67–77.
- Rajlich, V. and S. Raganathan (1998), "A Case Study of Evolution in Object Oriented and Heterogenous Architectures," *Journal of Systems and Software* 43, 85–91.
- Rajlich, V. and J. Silva (1996), "Evolution and Reuse of Orthogonal Architectures," *IEEE Transactions on Software Engineering* 22, 4, 153–157.
- Shaw, M. and D. Garland (1996), *Software Architecture*, Prentice-Hall, Upper Saddle River, NJ.
- Tichy, W. (1994), *Configuration Management*, Wiley, New York.
- Yau, S.S., J.S. Collofello, and T. MacGregor (1978), "Ripple Effect Analysis of Software Maintenance," In *Proceedings of Compsac*, IEEE Computer Society Press, Los Alamitos, CA, pp. 60–65.