A Methodology for Incremental Changes

Václav Rajlich
Computer Science Department
Wayne State University
Detroit, MI 48202
1-313-577-5423
rajlich@cs.wayne.edu

ABSTRACT
Incremental change is one of the key principles of Extreme programming. This paper presents a methodology and a case study of incremental changes using a small application written in UML and Java. Domain concepts play a key role in this methodology.

Keywords
Incremental programming, Extreme programming, software evolution, domain concepts, change propagation

1 INTRODUCTION

Extreme programming is based, among several other things, on the fast delivery of small releases [1]. They provide valuable feedback about the direction and progress of the project and allow timely adjustments of the project's goals and schedules. By this method, Extreme programming substantially lessens the inherent risk in the project.

This paper deals with incremental change, which is the foundation of small releases. Incremental change introduces new functionality into the program. It is different from other kinds of changes that preserve, retract, or modify existing functionality.

The paper presents a methodology for incremental change. The methodology is aimed more at a novice rather than at an experienced programmer. It deals with domain concepts and their role in incremental changes.

Section 2 explains the methodology. Section 3 describes the case study. Section 4 summarizes the experience. Section 5 relates this work to other research, and section 6 contains conclusions and future work.

2 THE BASICS OF THE METHODOLOGY

Incremental change adds new functionality to the application. We observed that it implements one or several closely related domain concepts. For example, the Point-of-Sale application needs to deal with several forms of payment and hence there is an incremental change that introduces "credit cards", "checks", etc.

We also observed that the new concepts introduced in an incremental change are not entirely new. Usually they are already present in the code, though only in a primitive or implicit form. For example, before the introduction of credit cards and checks, the concept "pay" was already a part of the program. However, the concept was in a primitive form, allowing only cash, represented as just one number. The incremental change implements the respective concepts explicitly and fully. Hence the domain concepts play a substantial role in the selection and implementation of incremental changes.

The concepts that are dependent on each other must be implemented in the order of their dependency. For example, the concept "tax" is dependent on the concept "item" because different items may have different tax rates and tax without an item is meaningless. Therefore, the implementation of "item" must precede the implementation of "tax". If several concepts are mutually dependent, they must be implemented in the same incremental change.

Mutually independent concepts can be introduced in arbitrary order, but it is advisable to introduce them in the order of importance to the user. For example, in the Point-of-Sale program it is more important to deal correctly with taxes than to support several cashiers. An application with correct support for taxes is already usable in stores with one cashier. An opposite order of incremental changes would postpone the usability of the program.

Each incremental change follows the following sequence of steps:

1. Select the domain concepts to be introduced or further developed.
2. Select the test cases for the new concepts and implement the new concepts as new classes.
3. In the old code, find where the concepts are already present (often primitively or implicitly).
4. Refactor old code if the old concepts are delocalized and can be localized into fewer classes.
5. Change old classes that contain old concepts so that they properly interact with the new classes.
6. Propagate the change through the remaining old code as far as necessary.

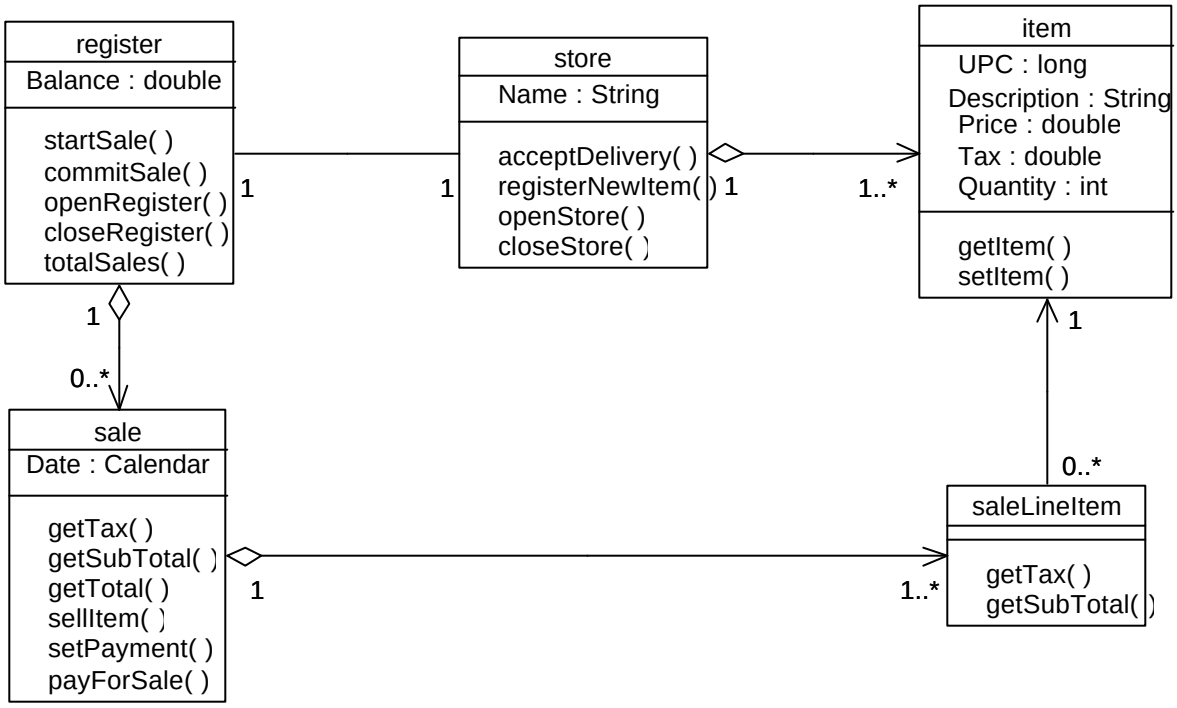

Figure 1

The methodology is illustrated by a case study of the next section.

3 CASE STUDY: POINT-OF-SALE

The case study deals with development of a Point-Of-Sale application that sells products in small stores. It uses UML in combination with Java.

The initial development started from scratch. The goal was to implement a working program that keeps an inventory of products, sells them, receives delivery, and supports a cash register. These concepts are closely tied together and must be present in any meaningful program. In keeping with the philosophy of Extreme programming, the classes were designed and implemented at a minimal level of functionality, see Figure 1.

In the second incremental change, we implemented support for credit cards and checks. We encapsulated all forms of payment in a base class *payment* and introduced subclasses for cash and authorized payments, with further subclasses for checks and credit cards. This class hierarchy was plugged into the original system through the interaction between the old class *sale* and the new class *payment*. Class *sale* was changed and a secondary change was done also in class *register*.

The third incremental change introduced support for price fluctuations. Products can have different prices at different times, for example limited time sales. We implemented this feature in classes *price* and *promoPrice*, which were plugged into the system through the interaction with class *item*. Class *item* was changed and the change propagated through old classes *store*, *saleLineItem*, and *sale*.

The fourth incremental change introduced the complexities of sale taxes. This incremental change is described in more detail in the Figures 2 through 6.

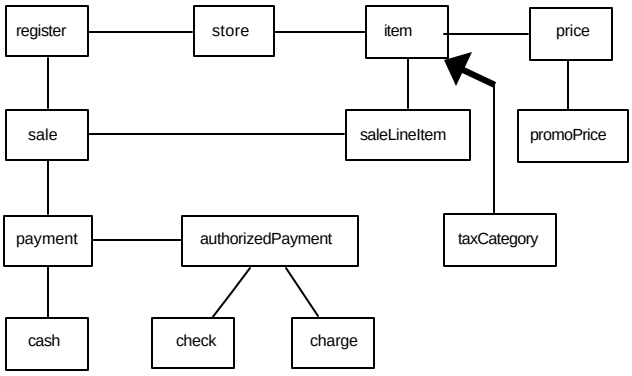

Figure 2

Different products may have different sales tax, depending on state law. This is done by a new class *taxCategory*, which is plugged into the program through interaction with class *item*, see Figure 2. The arrow points to the class that has to be fixed, in this case class *item*. (The old class *item* does not deal with *taxCategory* and must be changed to support this interaction.) The mark is denoted by bold arrow in the diagram, indicating the inconsistency and the direction in which it propagates. For simplicity, the diagram does not contain class

members or adornments. The notation is self-explanatory and it is based on [9].

In order to fix the inconsistency, the programmer visited class *item* and introduced a list of applicable taxes and relevant methods. These changes influenced classes *store*, *saleLineItem*, and *price*, see Figure 3. The visit to class *price* revealed that it does not need any changes and does not propagate the change further.

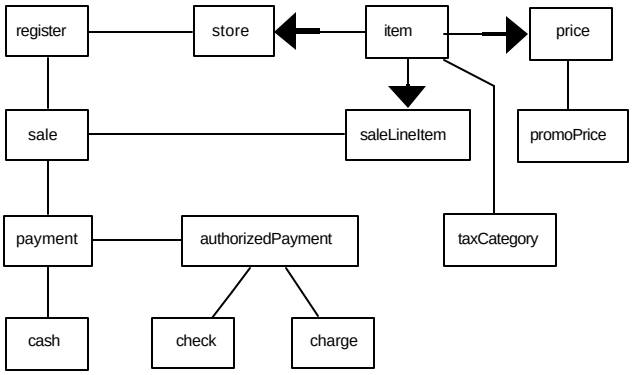

Figure 3

We then visited class *store* and modified it. The modified class can create new *taxCategory* instances for particular items. The next class, *register*, was visited but did not need any modification and the change did not propagate in that direction. The resulting diagram is in Figure 4.

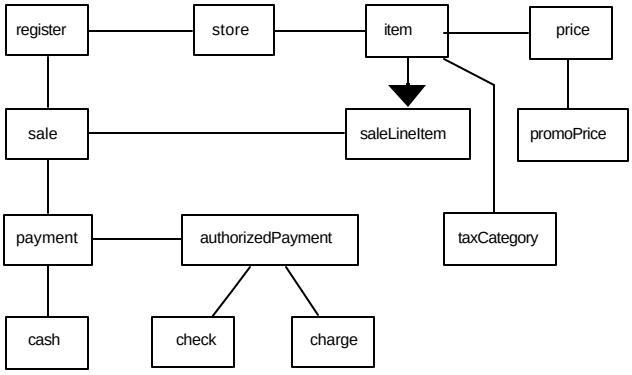

Figure 4

The next class we visited was *saleLineItem.* After the changes were performed, class *sale* could be influenced by the changes. Thus class *sale* is marked; see Figure 5.

Class *sale* required only one small change. Still, because it interacts with two other neighboring classes, there is a possibility that they may need a change, see Figure 6.

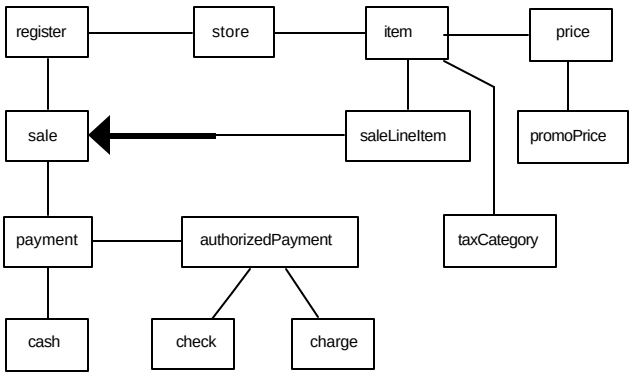

Figure 5

Class *payment* did not have to be changed and the change did not propagate further. Class *register* did not use the tax information contained in the *sale* class and relies upon prices supplied by the *sale* class. Therefore, there is no change in *register* and we unmarked it, without propagating the change further. This completed the fourth incremental change.

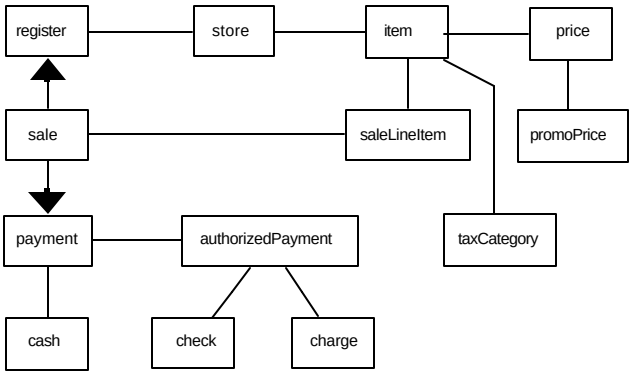

Figure 6

4 THE EXPERIENCE

The incremental changes of the case study introduced or substantially refined concepts of the application domain. We experimented with the size of the incremental changes and observed that excessively small changes usually do not decrease the number of classes to be visited. They lead to multiple class visits and the result is decreased efficiency of the work along with an increased likelihood of errors.

On the other hand, changes that are too big may overload the cognitive capabilities of the programmers, because the programmers must deal with too many issues at once. Hence, there is an optimal size of the incremental change. In this case study, all incremental changes introduced one to five new classes into the program, visited between three and seven old classes, and modified up to four of them.

During the change propagation, when a class was visited but not changed, should its neighbors have been visited also? There are several factors that influence this

decision. To miss possible change propagation means to introduce a subtle error into the code that will be hard to find later. Therefore we chose to be cautious, inspecting more classes than absolutely necessary. Still, the propagation must stop somewhere – otherwise every class of the program would be visited for every incremental change. The decision whether to propagate is based on properties of the underlying code. For example, it is more likely that a class that references the changed class will also need a change, rather than a class that is referenced by the changed class.

5 RELATION TO OTHER WORK

Extreme programming is described in [1]. In our view, it is a part of the life-cycle stage of software evolution, see [10].

When Extreme programming is used, it is often necessary to refactor already existing code [1], [8]. Authors of [5], [6], [7] study refactoring of C and C++ code. In this paper, we did not present refactoring change but recognize its importance.

In [9], the author defines evolving interoperation graphs that are used as a theoretical model of change propagation. They were used in Section 3 to describe change propagation related to one of the incremental changes.

There is much literature on software change during a later stage of software lifecycle called servicing [10] or legacy systems. During software servicing, the knowledge of software decreases and the emphasis of software change shifts towards program comprehension. The impact of a change on the program becomes an important issue. Change impact analysis is summarized in [2]. In [4] the authors describe how to locate concepts using computer-assisted search of software dependence graph.

6 CONCLUSIONS AND FUTURE WORK

In this paper, we presented a methodology and a case study of incremental changes. We observed the role domain concepts play in incremental changes.

Future work is to study the optimal size and optimal sequence of incremental changes depending on the relationship, coupling, and complexity of the domain concepts. The purpose is to minimize the change impact and rework during the change.

We are also planning to study changes that are not incremental, i.e. the changes that retract the functionality or merely modify it without adding significant new concepts. An issue is the relationship between refactoring and change propagation. The purpose of refactoring is to minimize the propagation. However, is it always possible to shorten the change propagation by refactoring, or are there certain changes that always delocalized and are "refactoring resistant"?

It is expected that the studies of incremental change will improve the current situation where the incremental change is largely a self-taught art, and will allow the accumulation of knowledge in this important field.

Acknowledgement

Milos Besta, a graduate student in Computer Science of Wayne State University, conducted the case study.

REFERENCES

1. Beck, K. Extreme programming explained. *Addison-Wesley*, 2000.
2. Bohner, S.A., Arnold, R.S. Software Change Impact Analysis. *IEEE Computer Society Press*, Los Alamitos, CA, 1996.
3. Booch, G., Rumbaugh, J., Jacobson, I. The Unified Modeling Language User Guide. *Addison-Wesley*, Reading, MA, 1998.
4. Chen, K., Rajlich, V., Case Study of Feature Location Using Dependency Graph. In *Proceedings of 8th International Workshop on Program Comprehension*, 2000, 241-249.
5. Fanta, R., Rajlich, V. Reengineering an Object Oriented Code. In *Proceedings of IEEE International Conference on Software Maintenance*, 1998, 238-246.
6. Fanta, R., Rajlich, V. Removing Clones from the Code. *Journal of Software Maintenance*, 1999, 223 - 243.
7. Fanta, R., Rajlich, V. Restructuring Legacy C Code into C++. In *Proceedings of IEEE International Conference on Software Maintenance*, 1999, 77-85.
8. Fowler, M. Refactoring Improving the Design of Existing Code. *Addison-Wesley*, Reading, MA, 1999.
9. Rajlich, V. Modeling Software Evolution by Evolving Interoperation Graphs, In *Annals of Software Engineering*, Vol. 9, 2000, 235-248.
10. Rajlich, V.T., Bennett, K.H. The staged model of the software lifecycle. *IEEE Computer*, July 2000, 66-71.