In M. Marchesi, G. Succi, D. Wells, L. Williams, Extreme Programming Perspectives,
Addison Wesley, 2003, 201 - 214

Chapter 19

A Methodology for Incremental Changes

Václav Rajlich

Incremental changes add new functionality and new properties to the software. This chapter presents a methodology for incremental changes, where domain concepts play a key role. A case study of a small application written in Java is also presented.

Introduction

Software evolution is a phase in the software life cycle in which major changes are made. It is based on iterative enhancements of the software [Rajlich+2000], called "small releases" in [Beck2000]. Each small release adds new functionality previously not available, and results in a working program. The strategy of small releases is to provide valuable feedback to programmers, managers, and customers and give them the opportunity to adjust future direction, goals, schedule, and budget [Beck2000].

Software methodologies have been used for a long time in initial development, which starts from scratch and ends with the first working version of software. There are a great variety of methodologies for initial development, supporting diverse software engineering processes and resulting in diverse software architectures.

In software evolution, the type of change that has been extensively studied and supported by methodologies and tools is software refactoring [Fowler1999; Fanta+1999]. Refactoring is a special kind of change that modifies the software's structure but does not affect the software's functionality. It is an important ingredient of software evolution because it allows programmers to restructure software and keep it understandable and evolvable. However, it is obvious that software evolution cannot consist of refactoring only; the thrust and purpose of evolution is to change software functionality and properties.

The change that modifies software functionality and other software properties is incremental change. It is the basic building block of software evolution, far surpassing in importance all other kinds of change. However, at this time, there are no integrated methodologies for incremental change. Instead, incremental change is largely a self-taught art that programmers learn by trial and error, sometimes at great cost. This chapter is a step toward a methodology and presents an outline of a methodology and a case study. It emphasizes the role of domain concepts in incremental change.

Role of Domain Concepts

The methodology presented in this chapter is based on the observation that requests for iterative change are formulated in terms of domain concepts, and these new concepts are already latently present in the code. For example, the Point Of Sale application needs to deal with several forms of payment, so there is an incremental change request to introduce "credit cards," "checks," and so on. However, even before this request, the concept "payment" was already in the program, represented as just one variable,

allowing only cash. The incremental change expands this latent concept and adds new functionality to it.

To make the change, the programmer locates the latent concepts and then reifies them, or implements them explicitly and fully. Based on this observation, we can formulate the steps of a methodology for incremental changes.

1. Formulate the change request.
2. Locate the latent concepts in the code.
3. Implement the concept explicitly and fully in new classes (concept reification).
4. Replace the latent concept in the old code with program dependencies (function calls, data flows, instance definitions) between the old and the new classes.
5. Propagate changes through the old code.

Let us look at these steps in more detail, using the previously mentioned change request to implement payments by credit card.

First, in the current program, the concept “payment” must be located. It can be a class, or it can be part of one or several classes. It can be implemented explicitly like a variable, or it can be a subset of values of a type, or it can be an implicit assumption that guided an implementation of an algorithm or data structure. Once the concept is located, it becomes clear how hard or easy it will be to incorporate the requested change into the existing program. The changes in concepts that are localized—that is, that are implemented in one or few classes—are easy to incorporate. On the other hand, the changes in concepts that are delocalized are hard. In the case of the credit card payments, the concept is implemented explicitly as the variable `payment` inside the class `sale`.

After the old concept has been located, the next step is to implement the concept fully with new functionality. In our case, the new implementation consists of a hierarchy of classes consisting of the base class `payment` and derived classes for cash and authorized payments, with further derived classes for credit cards and checks. This new hierarchy is then plugged into the old program and replaces the old implementation of the concept. This means that in the program, all references to the old concept must be replaced with references to this new hierarchy. Thus the old variable `payment` is replaced with an instance of the new class `payment`. All references to the old variable must be replaced with calls of the methods of this new class.

After that, the old relationship of class `sale` with its neighbors may be broken because of the changes in the class. Thus the programmer must check all its neighbors to see whether the relationship with the class `sale` has been affected. If not, there is no need to make any changes. However, if the relationship has been affected, it requires secondary changes in the neighbors. After these changes, the neighbors of the newly changed classes must also be checked, and so forth. This iterative process is called *change propagation* and continues until all real or potential inconsistencies are removed.

Concept location has been extensively studied; for example, see [Bohner+1996] and [Chen+2000]. Change propagation has also been studied; for example, see [Rajlich2000]. The case study in the next section contains examples of the methodology.

Case Study

The case study deals with a Point Of Sale application for small stores. The application keeps an inventory, receives deliveries, supports sales of items to customers, and

supports cashiers and cash registers. It is implemented in Java through several incremental changes.

Initial Development

The initial development implemented a greatly simplified version of the Point Of Sale application, in which all payments are in cash, prices are constant, taxes are uniform, and there is one register and one cashier in the store. Initial development introduced five classes: `register`, `store`, `item`, `sale`, and `saleLineItem`.

Support for Payment

Support for various forms of payment, including credit cards and checks, was implemented in the next iteration. Following the methodology, we first located the latent concept “payment” in the class `sale`, where it was implemented as a local variable of type integer. The member functions `setPayment()` and `payForSale()` deal with this variable and support cash payments. We reified the concept “payment” in a new class hierarchy consisting of the base class `payment` and its subclasses for cash and authorized payments, with further subclasses for checks and credit cards. This class hierarchy was integrated into the original program by replacing the old variable with an instance of the class `payment`. Other changes in the class `sale` involved changes in the member functions `setPayment()` and `payForSale()`. A secondary change propagated to the class `register`.

Price Fluctuations

This incremental change introduced price fluctuations. Products can have different prices at different times. In the old program, price was implemented as a single variable

in the class `item`. We implemented the new classes `price` and `promoPrice`, and replaced the variable in the old class `item` with an instance of the class `price`. After additional changes to the class `item`, the change propagated through the old classes `store`, `saleLineItem`, and `sale`. Figure 19.1 shows a Uniform Modeling Language (UML) diagram [Booch+1998] of the program at this stage.

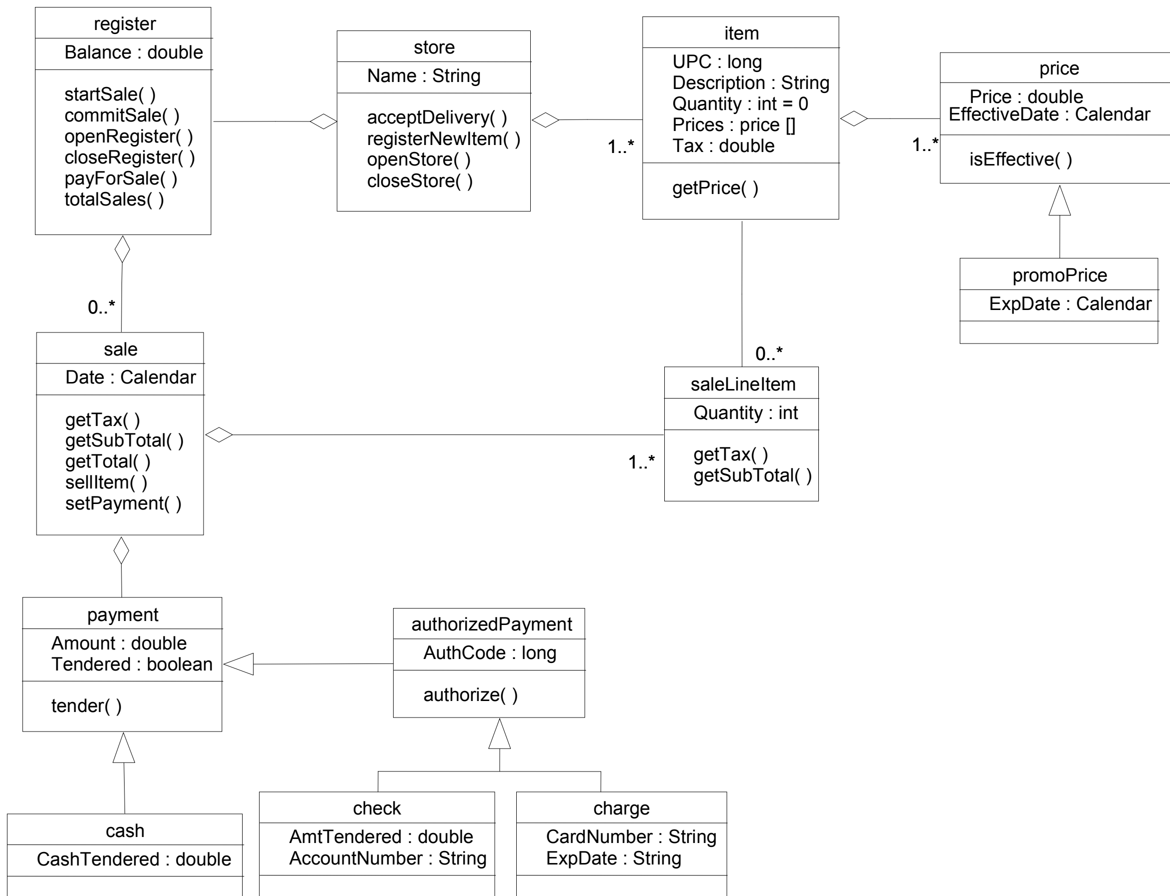

Figure 19.1: UML diagram of the Point Of Sale application

Sales Taxes

This incremental change introduced the complexities of sales taxes. We will describe this change in more detail with the help of the UML-like class diagrams shown in Figures 19.2 through 19.6. The boxes stand for classes, and the lines represent all

associations. The arrows indicate the inconsistencies in the program and point to the classes that have to be changed. See [Rajlich2000] for further details.

Different products may have a different sales tax, depending on state law. The old program contained a single tax rate as an integer within the class `item`. We implemented a new class, `taxCategory`, and then replaced the integer in the class `item` with an instance of `taxCategory`. The current state of the program is represented in Figure 19.2. In it, the arrow points to the class `item`, where several additional changes must be made.

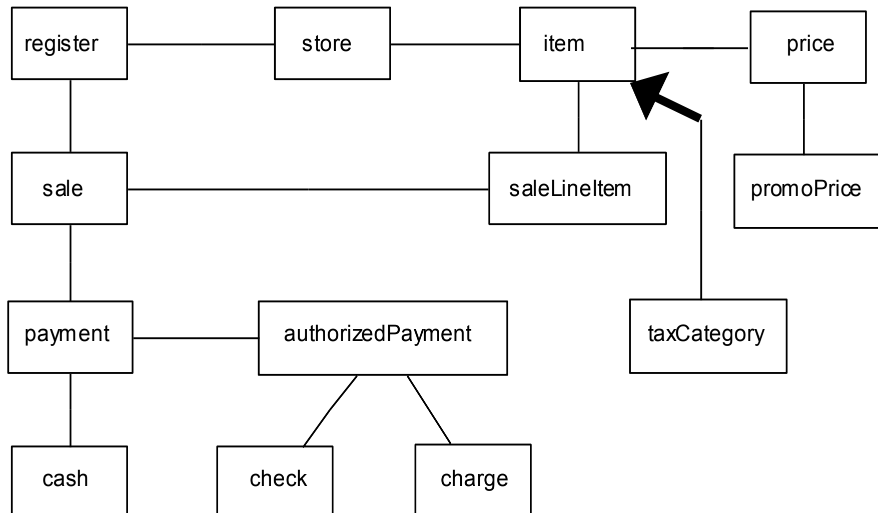

Figure 19.2: Point Of Sale application after implementing the class `taxCategory`

To remove the inconsistency, we examined the class `item` and made all necessary changes. The class `item` interacts with the classes `store`, `saleLineItem`, and `price`. Thus, they have to be checked and changed if necessary, as shown in Figure 19.3. The check of the class `price` revealed that it does not need any changes and that it does not propagate the change further. We then examined the class `store` and modified it. The

next class, `register`, was checked but did not need any modification and the change did not propagate in that direction. The resulting diagram is shown in Figure 19.4.

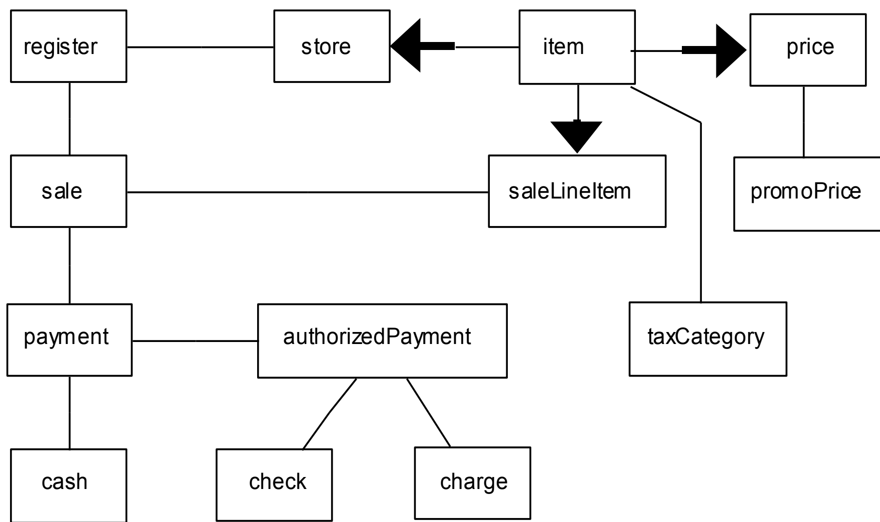

Figure 19.3: Examining classes that interact with the class `item`

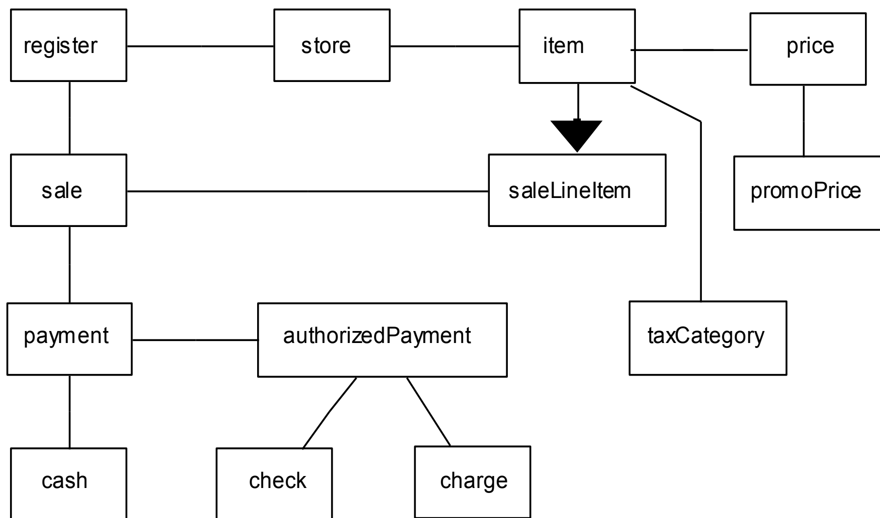

Figure 19.4: Exemining class `saleLineItem`

The next class we checked was `saleLineItem`. After the changes were performed, the neighboring class `sale` could be influenced by the changes. Thus the class `sale` is marked, as shown in Figure 19.5.

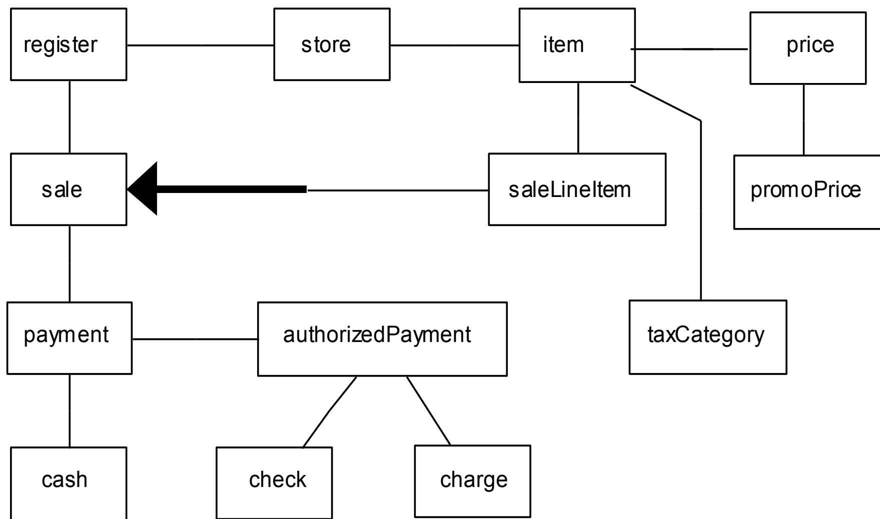

Figure 19.5: Marking the class `sale` for examination

The class `sale` required only one small change. Still, it interacts with two other neighboring classes as shown in Figure 19.6, and they may also need a change.

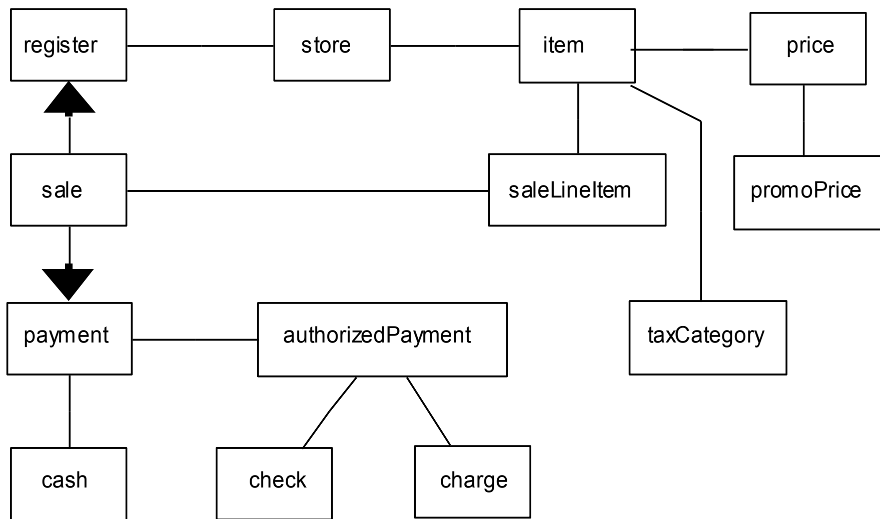

Figure 19.6: Exemining classes that interact with the class `sale`

The class `payment` did not have to be changed, and the change did not propagate further. The class `register` did not use the tax information contained in the `sale` class

and relied on prices supplied by the `sale` class. Therefore, no change was needed, so we unmarked it, without propagating the change further. That completed this incremental change.

Several Cashiers

Stores often have several cashiers who take shifts at registers. This incremental change introduces multiple registers, cashiers, and their sales. The old version of the application assumes one cashier and one register, and this assumption is implicit in the implementation of the classes `store`, `register`, and `sale`.

New classes `cashier` and `session` reify information about the start and the end of the cashier's sessions, about sales realized within each session, and about the specific register used by the cashier during the session. They are interconnected with the old program by using the classes `sale`, `register`, and `store`, as shown in Figure 19.7.

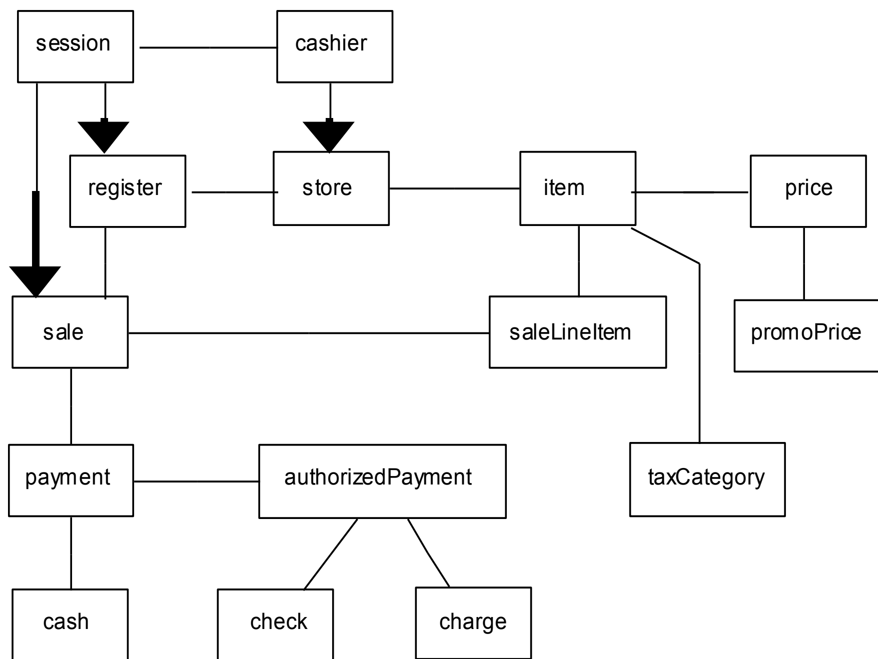

Figure 19.7: Interconnecting the new classes to the old program

Then we examined and modified the old class `register`, adding support for several sessions on the register and the cashiers' PIN numbers. After the changes, the class `register` no longer directly references the class `sale` but references the class `session` instead. A check of the class `sale` and its neighbors showed that the change does not propagate any further, as shown in Figure 19.8.

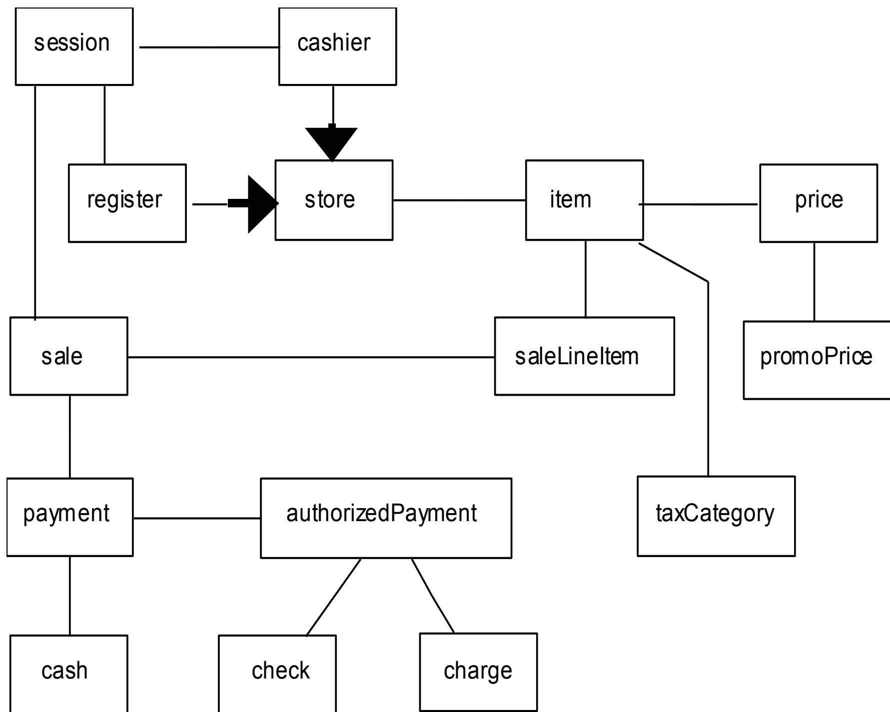

Figure 19.8: Exemining class `store` The last remaining class to examine is store. During the check, we propagated changes from both the class `cashier` and the class `register`. The changes include adding new data members for a list of cashiers and a new method, `registerCashier`, as well as modifying the methods `openStore` and `closeStore` to allow the store to open only when some cashiers are available. The check of the neighboring class `item` showed that there is no need to propagate the change further.

Observations

In the case study, we observed that both concept location and change propagation are an overhead of the change, and the overhead is the same whether the change is large or small. For example, if we introduced just credit card payments in an incremental change without also dealing with payments by check, the same concept location and change propagation would have to be done to implement that smaller change. Small changes lead to the repetition of the overhead tasks and result in decreased efficiency and increased likelihood of errors.

On the other hand, changes that are too big and deal with too many issues at once may overload the cognitive capabilities of the programmers and again lead to increasing the likelihood of error. Thus, 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, examined three to seven old classes, and modified up to four of them.

As we stated earlier, change propagation is a process that requires a check of all the neighbors of a changed class. To miss a change propagation means to introduce a subtle error into the code that will be hard to find later [Yu+2001]. Therefore, in our case study, we chose to be cautious and inspect more classes than absolutely necessary. However, there is a practical limit to how far to carry out change propagation; otherwise, every class of the program would be checked for every incremental change. In our case study, we used the following criterion: We inspected all classes that are direct neighbors of changed classes. Also, we inspected all classes that participate in a data flow with a changed class and checked whether the meaning of the data changed.

If so, we made the appropriate changes. If not, we no longer propagated the change. This criterion worked well in all instances in our case study.

Conclusions and Future Work

In this chapter, we presented an outline of a methodology and a case study of incremental change. We observed that domain concepts play an important role in incremental changes.

An interesting question is the optimal sequence of incremental changes that minimizes the changes' impact and any rework needed. Certain concepts depend on others, such as the concept "tax" depending on "item." In that case, the concept "item" must be introduced first, because without it, the concept "tax" is meaningless.

Another interesting question is the relationship between refactoring and change propagation. One of the purposes of refactoring is to minimize the propagation by localizing concepts in one or few classes. However, it is not clear whether it is always possible to shorten change propagation by refactoring. There may be certain changes that are always delocalized and are "refactoring resistant." Additional case studies are needed to answer this question.

A hope is that the methodology of incremental change will improve the current situation, in which the incremental change is largely a self-taught art. This will enable the accumulation of knowledge in this important field.

References

[Beck2000] K. Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.

[Bohner+1996] S. Bohner, R. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, 1996.

[Booch+1998] G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.

[Chen+2000] K. Chen, V. Rajlich. "Case Study of Feature Location Using Dependency Graph." *Proceedings of 8th International Workshop on Program Comprehension.* IEEE Computer Society Press, 2000.

[Fanta+1999] R. Fanta, V. Rajlich. "Removing Clones from the Code." *Journal of Software Maintenance*, 1999.

[Fowler1999] M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Rajlich2000] V. Rajlich. "Modeling Software Evolution by Evolving Interoperation Graphs." *Annals of Software Engineering*, Volume 9, 2000.

[Rajlich+2000] V. Rajlich, K. Bennett. "The Staged Model of the Software Lifecycle." *IEEE Computer*, July 2000.

[Yu+2001] Z. Yu, V. Rajlich. "Hidden Dependencies in Program Comprehension and Change Propagation." *Proceedings of 9th International Workshop on Program Comprehension.* IEEE Computer Society Press, 2001.

About the Author

Václav Rajlich is a full professor and former chair in the Department of Computer Science at Wayne State University. He has published extensively in the areas of

software evolution, maintenance, change, and comprehension. He received a Ph.D. in mathematics from Case Western Reserve University. Contact him at vtr@cs.wayne.edu.