

The Role of Incremental Change in Agile Software Processes

Neal Febbraro, Václav Rajlich

Department of Computer Science, Wayne State University, Detroit, MI 48202

nfebbrar@ford.com, rajlich@wayne.edu

Abstract

This paper presents a model of incremental change that consists of concept location, impact analysis, actualization, change propagation, supporting refactorings, and testing. Repeated incremental change is the foundation of an agile process called Concept-based Incremental Development (CID). A case study of a Point of Sale system illustrates the usefulness of CID. The paper argues that CID complements those agile processes that concentrate on team and management issues, and therefore can be combined with them.

1. Introduction

Agile processes build software starting with a greatly simplified but functioning first version, and iteratively adding new features to it. Incremental Change (IC) is the process of adding new functionality to existing code, and it is a basic component of all agile processes.

In most of the current literature, IC is mentioned only briefly, and it is assumed that it is a well-known process that does not need any extensive explanation. This contrasts sharply with the other technical component of agile processes, refactoring, that has received considerable attention from both researchers and practitioners. In this paper, we aim to address this imbalance and present a detailed model of the IC process. This can be considered as a step towards the creation of a body of knowledge for IC that is similar in scope and detail to the body of knowledge that refactoring already has.

In order to demonstrate the usefulness of our model of IC, we define an agile process for developing software that consists of repeated IC; we call it “Concept-based Incremental Development” (CID). This process builds software in functional increments, and we show that it promotes the values of agile software development. Since it focuses only on the

details of IC, it can be combined with those agile processes that focus on managerial or team issues.

Section 2 of the paper contains an overview of the related work. Our model of IC and the description of CID are presented in section 3. Section 4 presents a case study in which CID is used to develop a Point of Sale system. Section 5 presents conclusions and future work. The appendices present details of two steps of CID in the Point of Sale case study.

2. Previous Work

Much of the research regarding incremental change has been done in the context of software maintenance. An early model of IC during software maintenance is presented in [1].

More recently, the IC process is explored by Rajlich and Gosavi in [2]. Their case study adds a new feature into the open source software framework *Drawlets*. The paper focuses on the IC activities of concept location, change propagation, and refactoring. A follow-up case study by Skoglund and Runeson [3] presents testing for each of the three change strategies implemented in the original case study. It also describes the testware and techniques that keep the regression tests up to date.

Individual activities of IC have been a subject of numerous research papers. Concept location finds where a domain-specific concept is implemented in code, which can be a difficult issue in a large, unfamiliar application. Static techniques for concept location, which include pattern matching using *grep*, class dependency search, and information retrieval approach, are detailed in [4]. JRipples is an Eclipse plug-in that assists in concept location using static class dependency search [5]. A dynamic approach called software reconnaissance is explored in [6]; it analyzes execution traces in order to locate feature implementation in the code.

Impact analysis determines the potential impact that a change might have on the software system. It detects

the classes that will most likely be affected by a change, and helps to determine a strategy for IC implementation. A classification of impact analysis approaches is given in [7]. The JRipples tool described in [5] assists the programmer in performing impact analysis by building a class dependency graph and marking classes that are potentially impacted by the change. The programmer then visits these classes and determines whether they will actually change.

Refactoring is defined as a “change made to the internal structure of the software to make it easier to understand and cheaper to modify without changing observable behavior” [8]. Refactoring can be done to shorten the change propagation in software, as was done in [2]. A review of the recent state of the art in refactoring processes is given in [9]. This review also points out criteria that need to be taken into account when building tool support for refactoring. Using refactoring tools lessens the chance that a refactoring is done incorrectly and introduces an error. An extensive list of refactoring tools is given in [10].

Unit testing determines whether a refactoring has been done correctly, or if a given change has introduced new bugs in the system. It involves keeping a suite of unit tests that can be run automatically. The JUnit framework is an example of a unit testing utility for Java programs that will automate the unit testing process [11]. The Abbot framework automates testing of a user interface using scripts [12]. The case study done by Skoglund and Runeson in [3] used both of these tools to automate the unit and functional tests for *Drawlets*.

Test-driven development [13] creates unit tests before the actual code is written. This assists the programmer in writing simple code that produces only the functionality that is expressed in the test cases. As an important side-product, this technique creates an up-to-date suite of unit tests that can be used in future regression testing. The code is then refactored to improve the architecture.

The Agile Manifesto [14] characterizes agile processes and states that programmers should “...welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.” According to a review of agile processes in [15], “Agile proponents ... concentrate only on the functions needed immediately, delivering them fast, collecting feedback, and reacting rapidly to business and technology changes.”

Scrum [16] is an example of an agile software process that is motivated by changing requirements in a volatile environment. Development is done in

increments called sprints, and the program architecture evolves based on this incremental development rather than an upfront design. Daily meetings between members of a sprint team help address emerging issues quickly. Larger meetings held after each sprint amongst all sprint teams help share knowledge and experience among the entire group, and also help to identify any issues that could potentially impede future progress.

Extreme Programming [17] is one of the most widely known agile processes. It focuses on the core values of simplicity, communication, feedback, courage, and respect. Extreme programming advocates believe that changes in requirements and environment are an inevitable part of software development; developers should embrace the change, and adapt to the requirements change rather than trying to predict it. Twelve specific practices are defined, and they promote the five core values.

3. Incremental Change and Concept-based Incremental Development

Incremental change is the foundation of software maintenance, evolution, and agile processes. In this section we present a model of IC that is the result of our experience and is based on case studies performed in [2] and [3]. It was also successfully used in undergraduate and graduate software engineering course projects [18]. IC process is presented as a set of activities where each activity plays a specific role.

3.1. Activities of Incremental Change

The following are the activities of an IC:

- *Initiation*: Analyze the user story or change request and extract relevant concepts.
- *Concept Location*: Locate the concepts in the code.
- *Impact Analysis*: Identify a complete set of classes likely to be affected by this incremental change.
- *Prefactoring*: Refactor to make the change easier.
- *Actualization*: Implement the concepts by writing new code, and then incorporate this new code into the existing system.
- *Change Propagation*: Propagate the change through the system and correct inconsistencies.

- *Postfactoring*: Refactor to eliminate any design flaws (“bad smells”) introduced during the change.
- *New Baseline*: Commit the new code to the code repository, update the software documentation, and release the new version to the customers.

Testing is done throughout the IC process. Regression testing is used after both prefactoring and postfactoring to ensure that these activities have not introduced errors into the system. Test cases are created as part of the actualization activity in order to test the new functionality and to update the test suite.

Note that the process is a guideline and some activities may not be needed or may be trivial for some IC’s. For example, impact analysis is trivial in a very small system early in development. A programmer who is familiar with a software system will not need to spend much time performing concept location. In circumstances such as these, some activities can be done quickly or skipped altogether.

The activities are described in more detail in the rest of this section.

3.2. Initiation

An IC is initiated by a request to add new functionality to the software system. This functionality can be found in user stories, change requests, or requirements documents. These sources are often written in natural language and use the terminology of the problem domain. The programmer analyzes these written sources and extracts the relevant concepts.

3.3. Concept Location

The concepts extracted during IC initiation are used as a starting point of concept location. We identify two types of concepts: *explicit* and *implicit*.

Explicit concepts are directly represented in the code as fields, methods, code snippets, and/or classes. For example, the document page count of a word processing program is explicitly present in the code as an integer. Since this concept is present in the code, it can be found by the programmer.

Implicit concepts, on the other hand, are implied by the code, but not expressed. Using the same word processor example, an implicit concept would be authorization to open files. It is implied by the software that any user is authorized to access any word processor file. Implicit concepts cannot be directly located since they are not implemented as code; the

programmer must find similar concepts that point to the correct place where the new concept will be implemented.

Several techniques have been researched to aid the programmer in this task, including both static and dynamic program analysis [4, 6].

3.4. Impact Analysis

Impact analysis is a process by which the programmer analyzes the software system to determine what software components will be affected by an IC. The class or classes produced by concept location are called the *initial impact set*. Classes that are neighbors to the classes in the initial impact set are inspected in order to determine if they will also need to change due to the ripple effect. If a class will need to change, it is added to the set. The process is then repeated for this class, until it is determined that a change will not propagate any further. The final set of impacted classes is called the *impact set*.

While the common avenue of performing impact analysis is to look at direct dependencies between classes, there are certain types of dependencies that are not as obvious. Called *hidden dependencies*, they represent data flows between classes that are not directly associated with each other. An example of hidden dependencies is given in [19].

The information gained from impact analysis can help programmers to plan how they are going to implement an IC, and allow them to explore various options.

3.5. Prefactoring

Prefactoring is an opportunistic refactoring that helps to localize IC. If the impact set becomes large, a prefactoring may limit its size and hence shorten change propagation. An example of refactoring used to limit the impact of a change in a software system is shown in [2].

Also, if an explicit concept does not have its own class, it can be localized by the *extract class* refactoring [8] that gathers relevant fields, methods, and/or code snippets into a new class. The programmer then expands this class to include new functionalities outlined in the change request.

3.6. Actualization

Actualization implements the new aspects of the concept. If the concept is implicit, then actualization creates a new class containing the new code of the concept. The new class is then plugged into the rest of the code by creating instances of this new class in the places indicated by concept location. This creates program dependencies between the new class and the rest of the system.

For an explicit concept, there is already code containing the concept that may have been extracted into a specific class during the prefactoring. That class has to be modified based on the change request. It is already connected to the rest of the program through already existing program dependencies, but additional class dependencies may be needed.

3.7. Change Propagation

If a class changes, other classes that are associated with it may also need to change. The change propagation activity ensures that a change made in one class is propagated properly throughout the entire system. This ensures that the code remains consistent with respect to changes made during actualization of the concept.

Starting with the changed class, the programmer follows the class dependencies and visits all neighbor classes to see if they need to be changed. If a class does change, its set of neighbors will now have to be visited as well. This process continues until the change has been fully propagated through the system.

Depending on circumstances, change propagation can be simple or complex. Changes to a class's public interface will obviously propagate to any other class that uses this interface. Other changes are more subtle; for example, a method that returns the total cost for a set of items is changed so that it also includes tax in the final price. The type of output and interface of the method has not changed, but the semantics have changed and will need to be taken into account. Because of this, all classes interacting with a changed class should be inspected during change propagation [2].

3.8. Postfactoring

During postfactoring, the programmer examines the code and determines if any anti-patterns have been introduced as a result of the IC. These anti-patterns have been called "bad smells"; they make the software

more difficult to comprehend and harder to change in the future [8]. Postfactoring eliminates these bad smells and the resulting improved software architecture makes future incremental changes easier.

3.9. New Baseline

After the completion of an IC, the code can be checked into the version control system. This produces a new baseline to work from for future IC. It may take several rounds of IC before a user story or a requirement is fully implemented. In this case, the IC process is repeated until the feature is completed in full. After the feature is implemented, a new version can be released to the customers.

This is the time when the programmer can use knowledge gained from working with the software and update the software documentation. This would include code-level comments, external code documentation, user manuals, and so forth. Research has shown that external documentation updates can be done relatively inexpensively with the appropriate tool support, and can have a positive impact on project economics [20].

3.10. Testing and IC

Testing is done as part of prefactoring, actualization, and postfactoring. Fowler states that an essential precondition for refactoring is having a suite of solid, automated unit tests [8].

The *extract class* prefactoring does not break existing testware [21] and therefore the existing regression test suite can be used. However, some methods may have been moved from the old class to the extracted class and the test cases associated with these methods should also be moved from the old test class to the new one.

The test suite is also important in the postfactoring activity, but some of these refactorings may invalidate unit tests; techniques have been proposed to solve this issue [21].

As part of the actualization activity, a new test class is created that will contain test cases for the new functionality. A programmer can choose, for example, a test-first approach and build the test cases before creating new methods.

Functional tests for the new functionality are created during actualization. For software that includes a graphical user interface, a tool such as Abbot [12] can help automate the tests. These functional tests ensure that the new feature is implemented correctly.

3.11. Concept-based Incremental Development

In order to illustrate the usefulness of our IC model, we defined a development process called “Concept-based Incremental Development” (CID) that is based on repeated IC. It begins with a very simple implementation that includes a small subset of core functionality.

Features that come from various sources such as user stories or requirements documents are broken into well-defined functional increments based on relevant domain concepts, and each of the increments is implemented by a specific IC. The functionality is added only on an as-needed basis, and CID does not try to predict future additions or changes. Each IC is called an *iteration* or *step* of CID.

CID concentrates on the technical issues of how to incrementally add additional functionality to existing code. Other agile processes, such as Scrum [16], focus on organizational issues of software development. CID presents a complementary view, and processes such as Scrum can be combined with it to provide a more complete agile solution.

4. Case Study

The goal of the case study is to illustrate CID on a well-understood example, and to compare the results with another method, Object-Oriented Design [22]. The application is written in Java and the JUnit framework is used for the automated unit testing.

4.1. The Point of Sale Application

We developed a Point of Sale application and studied the resulting structure of the software. The following requirements are used to develop the application:

1. The system keeps track of an inventory of items by UPC, item name, price, tax, and current quantity available.
2. Allow cashiers to log in, and keep track of sales made during a cashier session. A session involves the time between the cashier logging in and out, and records of sales made during this time.
3. Sale prices can also be included that will always override a regular price.

4. Support a customer receipt containing one or more items, as well as data such as date of sale and processing cashier.
5. Acceptable payments include cash, credit, and check.
6. Keep track of total store cash balance.

The initial very simple version keeps track of the price, tax, and quantity of a single item, keeps track of the store cash balance, and allows the cash-only sale of items. This is implemented as a single class *Store*.

During each additional iteration, a new feature is added to the system using the IC process of section 3. The sequence of functional increments is:

- 1) Initial version.
- 2) Expand inventory to support multiple items.
- 3) Support multiple prices with effective dates.
- 4) Implement promotional prices.
- 5) Support the log- in of a single cashier.
- 6) Support multiple cashiers.
- 7) Add cashier session.
- 8) Keep detailed sale records such as item sold and date/time of sale.
- 9) Support multiple items per transaction.
- 10) Expand concept of cash payment to include cash tendered, change, and keep track of these values with regards to a specific sale.
- 11) Implement credit card payment.
- 12) Implement check payment.

We present steps 5 and 7 in greater detail in appendix A and B, as they illustrate adding an implicit concept to the code and expanding an explicit concept, respectively. Figure 1 depicts the final architecture of the Point of Sale application at the end of the case study.

4.2. Results and Discussion

We made several observations during the case study. Actualizing an implicit concept always created a new class where all code implementing that functionality resides. Explicit concepts were extracted into a new class from primitive code snippets already in the code.

One of the resulting anti-patterns we observed in the case study is *data class*. This is a situation where a class contains nothing but fields and get or set methods for them. If other classes are performing operations on these fields, the *extract method* and *move method*

refactorings [8] move these methods to the data class in order to reduce coupling. This was done with the *Sale* class in step 8. This class has fields containing the sale subtotal and total; the two values were originally calculated in class *Store* and the refactoring moved the respective methods into the *Sale* class.

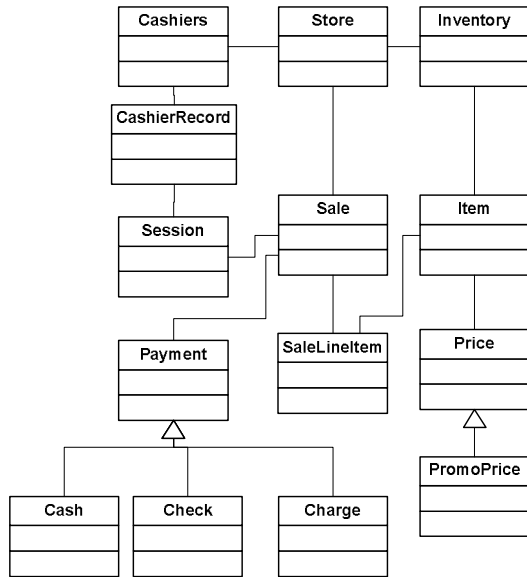


Figure 1. Point of sale system produced by concept-based incremental development

Another bad smell that came up several times is *long method*. Adding features sometimes caused methods to expand beyond acceptable limits. An example of this was when new functionality of cashier performance needed several new nested loops that printed individual cashier session data. We solved that using *extract method* refactoring.

Another bad smell that can arise is *middle man* defined by Fowler [8]. A class is considered to be a middle man when it is doing a lot of simple delegation. This can happen when a class is stripped of its functionality as a consequence of *extract class* prefactoring; if a class is repeatedly extracted from, it can end up doing nothing other than transferring information between other classes. Fowler describes the *remove middle man* refactoring [8] to deal with this situation. However the optimal amount of delegation is debatable, because some situations may require more and others less of it. In this case study, we deliberated and decided to leave all delegations in place in order to hide implementation details between classes.

Table 1 shows the growth of the suite of unit tests after each step. Each IC adds at least one test class to the system. Some IC's, such as step 6 (multiple cashiers), do not add many assertions because much of the functionality of class *Cashier* was extracted into class *CashierRecord*, and therefore the associated test cases were moved from *Cashier*'s test class to *CashierRecord*'s test class.

Table 1. Size of test suite during concept-based incremental development

| Step | Test Classes | Assertions |
|------|--------------|------------|
| 1 | 1 | 12 |
| 2 | 3 | 34 |
| 3 | 4 | 37 |
| 4 | 5 | 40 |
| 5 | 6 | 53 |
| 6 | 7 | 55 |
| 7 | 8 | 65 |
| 8 | 9 | 68 |
| 9 | 10 | 71 |
| 10 | 11 | 77 |
| 11 | 13 | 89 |
| 12 | 14 | 98 |

Figure 2 represents the Point of Sale design produced by Object-Oriented Design methods and published in chapter 1 of [22]. When comparing figure 1 and figure 2, there are significant similarities and differences. Both CID and Object-Oriented Design identified the same important concepts and encapsulated them as classes: item, sale, payment, price, promotional price, cashier, session, and store.

However the differences reflect the different philosophy of the two methodologies. Several classes in figure 2 are created in anticipation of future changes and additions; an example of this is the *UPC* class. This class is a wrapper for the value of an item's UPC code. In [22], it is explained that it is good to add classes such as this to the system even if they contain a single attribute so that future changes to UPC functionality would be done in this class. Under CID, the UPC concept will not be encapsulated as a separate

class until the programmers need to expand the functionality of the explicit concept UPC, and then it will be extracted. This keeps the design simpler because extra classes are not added to the system until the time when they are actually needed.

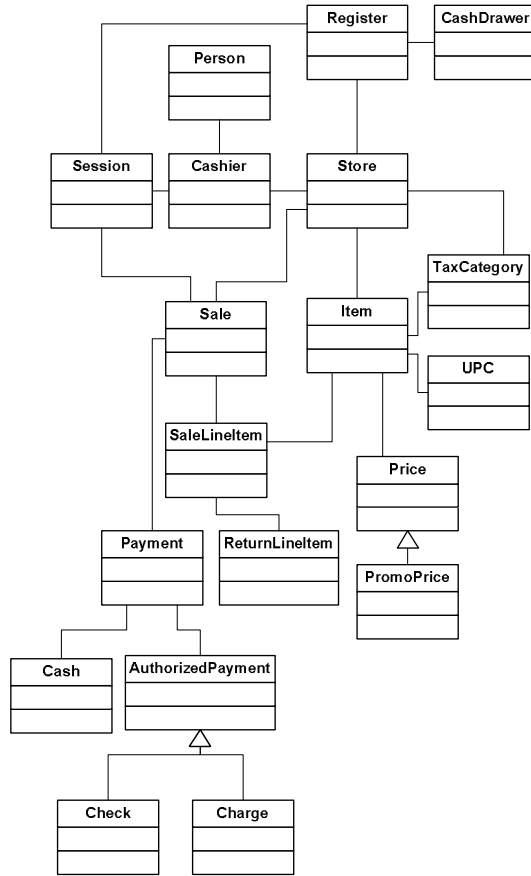


Figure 2. Point of sale system design based on object-oriented design

The question may arise whether CID belongs to the category of agile software developments. The case study demonstrated the following values of the Agile Manifesto [14]:

- Continuous delivery of working software
- Welcome changing requirements
- Deliver software frequently on a short timescale
- Simplicity

The remaining values of Agile Manifesto deal with the organizational and team issues and the case study

did not deal with those. However we are convinced that CID can be combined with other agile processes where the focus is on these issues.

5. Conclusions and Future Work

In this paper, we presented a model of Incremental Change (IC) and a development process that consists of repeated IC. We showed an example of a program developed in this way, and observed its properties. This process complements many other agile processes because it concentrates on incremental change, while other agile processes focus on organizational aspects of the software development.

The activities of IC include concept location, impact analysis, and change propagation. These activities can be facilitated by specialized software tools; at least one such tool is currently being developed.

Several specific refactorings (*extract class*, *remove middle man*) play a prominent role in IC. Because of this prominence, specialized refactoring tools are being developed specifically for these refactorings.

Additional case studies, where IC is used in combination with other agile processes, are planned for the future.

Acknowledgements

This research was supported in part by grants from the National Science Foundation (CCF-0438970), the National Institute for Health (NHGRI 1R01HG003491), and 2005 and 2006 IBM Faculty Award.

References

- [1] Yau, S., Nicholl, R., Tsai, J., Liu, S., "An Integrated Life-Cycle Model for Software Maintenance", *IEEE Transactions on Software Engineering*, Aug. 1988, vol. 14, issue 8, pp. 1128 - 1144.
- [2] Rajlich, V. and Gosavi, P., "A Case Study of Unanticipated Incremental Change," *Proceedings of the International Conference on Software Maintenance (ICSM'02)*, Oct. 3-6 2002, pp. 442-451.
- [3] Skoglund, M., Runeson, P. "A Case Study on Regression Test Suite Maintenance," *Proceedings of the 20th International Conferenc. on Software Maintenance*, Sept. 11-14 2004. pp. 438-442.

[4] Marcus, A., Rajlich, V., Buchta, J., Petrenko, M., Sergeyev, A., "Static Techniques for Concept Location in Object-Oriented Code," *Proceedings of the 13th International Workshop on Program Comprehension.*, May 15-16 2005, pp. 33-42.

[5] Buckner, J., Buchta, J., Petrenko, M., Rajlich, V., "JRipples: A Tool for Program Comprehension During Incremental Change," *Proceedings of the 13th International Workshop on Program Comprehension*, May 15-16 2005, pp. 149-152.

[6] Wilde, N., Scully, M. C., "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance*, vol. 7, issue 1, 1995, pp. 49-62.

[7] Arnold, R.S., Bohner, S.A., "Impact Analysis - Towards a Framework for Comparison," *Proceedings of the Conference on Software Maintenance 1993 (CSM-93).*, Sept. 27-30 1993, pp. 292-301.

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

[9] Mens, T., Tourwe, T., "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, issue 2, Feb. 2004, pp. 126-139.

[10] Fowler, M., *Refactoring* homepage <http://www.refactoring.com/tools.html>

[11] Gamma, E., Beck, K. JUnit framework for automating unit tests- <http://www.junit.org/>

[12] Wall, T. Abbot framework for automated functional testing. <http://abbot.sourceforge.net/>

[13] Janzen, D., Saiedian, H., "Test-driven Development Concepts, Taxonomy, and Future Direction." *Computer*. vol. 38, issue 9, Sept. 2005. pp. 43-50.

[14] The Agile Manifesto. <http://agilemanifesto.org/>

[15] Abrahamsson, P., Warsta, J., Siponen, M.T., Ronkainen, J., "New Directions on Agile Methods: A Comparative Analysis," *Proceedings of the 25th International Conference on Software Engineering*, May 3-10 2003, pp. 244-254.

[16] Schwaber, K., Beedle, M., *Agile Software Development with Scrum*. Prentice Hall, 2002.

[17] K. Beck, *Extreme Programming Explained*. Addison-Wesley, 1999.

[18] Buchta, J., Petrenko, M., Poshyvanyk, D., Rajlich, V., "Teaching Evolution of Open-Source Projects in Software Engineering Courses," *IEEE 22nd International Conference on Software Maintenance (ICSM '06)*, Sept. 2006, pp. 136 – 144.

[19] Zhifeng, Yu., Rajlich V., "Hidden Dependencies in Program Comprehension and Change Propagation," *Proceeding of the 9th International Workshop on Program Comprehension*, May 12-13 2001, pp. 293-299.

[20] Rostkowycz, A. J., Rajlich, V., "A Case Study on Long-Term Effects of Software Redocumentation.", *Proceedings of the 20th IEEE International Conference on Software Maintenance*. 11-14 Sept. 2004. pp. 92-101.

[21] Van Deursen, A., "Video Store Revisited: Thoughts on Refactoring and Testing," *XP 2002.*, May 2002.

[22] Coad, P., North, D., Mayfield, M. *Object Models: Strategies, Patterns, and Applications*. Prentice Hall, 1996.

Appendix A: Cashier Login

This step of Concept-based Incremental Development (CID) illustrates Incremental Change (IC) for an implicit concept.

A.1. Initiation

This IC is initiated by the following user story:

Allow cashiers to log in, and keep track of sales made during a cashier session. A session involves the time between the cashier login and logout, and records sales made during this time.

Analyzing this user story, the concepts present are *cashier* and *login*. Currently there is not any code that represents the concepts of cashier or login; the system assumes that anyone who runs the software is an authorized cashier. Therefore, the first activity is to take the implicit concept of *cashier* and locate it in the code.

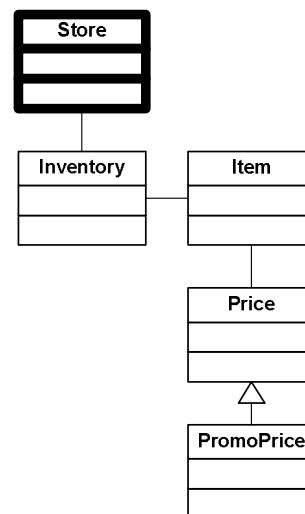


Figure 3. Concept location for cashier

A.2. Concept Location

Since the concept of *cashier* is not expressed as code, we located areas of the code that are somehow related to the cashier. The *Store* class is a likely candidate because it has fields for the store balance and contains an instance of *Inventory*, therefore it is a logical location to store cashier information as well. The situation is presented in figure 3.

A.3. Impact Analysis

Since the *Store* class will be changed, this is the starting point of impact analysis. Class *Inventory* is visited next, and determined that it will also need to change. After this, its neighbor class *Item* is visited, but will not change and does not propagate the change further. The impact set is shown in figure 4. Since the concept of cashier is implicit, there is no prefactoring in this IC.

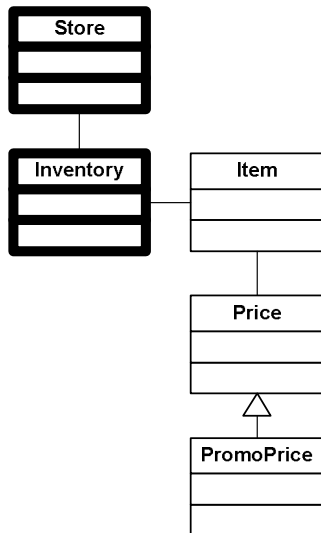


Figure 4. Impact set for addition of cashier

A.4. Actualization

During the actualization activity, a new class *Cashier* is created. The class includes fields for the name, cashier ID, password, and so forth. This new class is incorporated into the system by creating instance of the new class in class *Store*, as determined during concept location. This creates structural

dependencies between the new class and the rest of the software in figure 5.

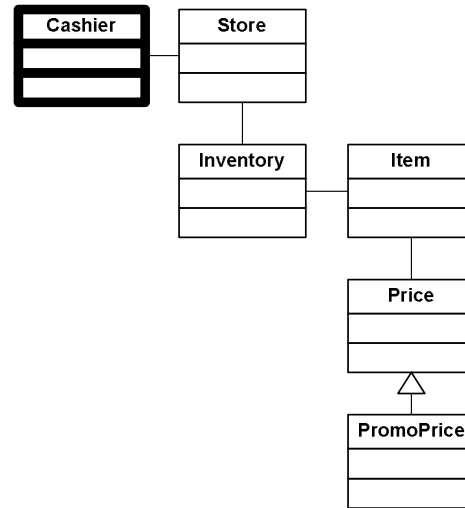


Figure 5. System after actualizing the *Cashier* class

A.5. Change Propagation

This step propagates the changes through the rest of the system. We begin in the class *Store* and visit all neighbors to see if they need to be changed due to our incremental change. The change needed to be propagated to class *Inventory*, see Figure 6.

In this step, no bad smells are created and hence no postfactoring is needed.

A.6. New baseline

After the completion of the IC, the new code is committed to the version control system. This creates a new baseline of the code base that future incremental changes will use.

A.7. Testing

A new test class for *Cashier* is created during the actualization activity, and assertions are created to test the functionality of the new methods. Since *extract class* was not used to create *Cashier*, test cases from other classes did not need to be moved. 13 new assertions are made to test *Cashier*.

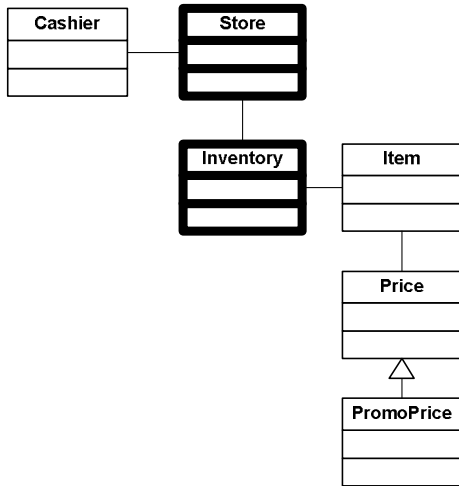


Figure 6. Visited classes during change propagation

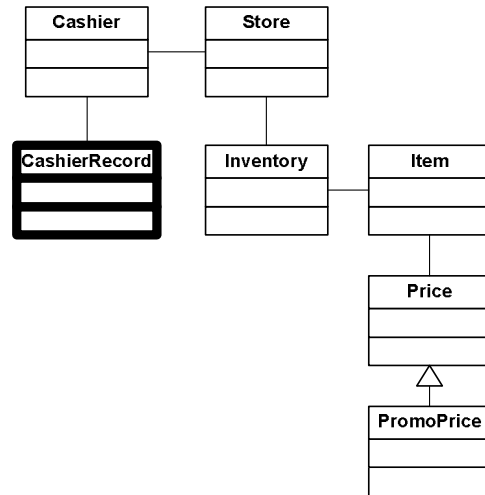


Figure 7. Location of session concept

Appendix B. Cashier sessions

This step of CID illustrates IC for explicit concept.

B.1. Initiation

This IC is initiated by the following user story:

Allow cashiers to log in, and keep track of sales made during a cashier session. A session involves the time between the cashier logging in and out, and records of sales made during this time.

The concept extracted from this user story is the concept *session*.

B.2. Concept Location

Using concept location, the explicit concept is located in class *CashierRecord* as indicated in Figure 7. This class contains snippets of cashier information such as name, id, and password. It also stores login dates and times for the cashier.

B.3. Impact Analysis

Impact analysis reveals several classes that will be affected by this change, as indicated by the classes highlighted in figure 8. These were identified by studying the class dependency diagram, and determining how far a change in *CashierRecord* would propagate.

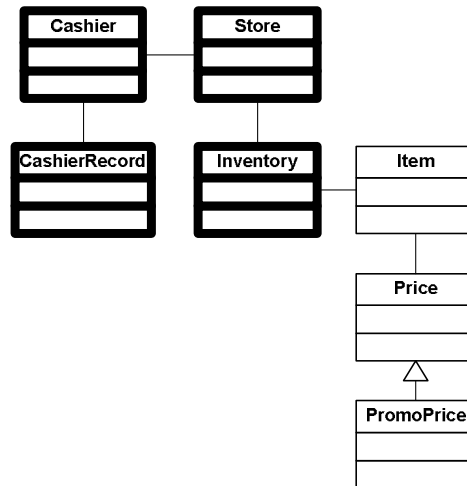


Figure 8. The impact set identified during impact analysis

B.4. Prefactoring

Extract class refactoring of relevant fields and code snippets in class *CashierRecord* yields a new class *Session*, see Figure 9.

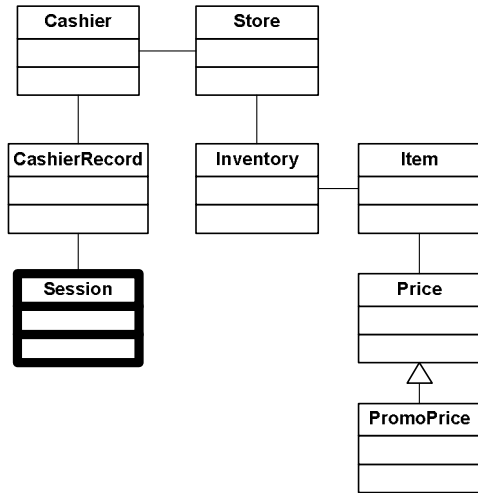


Figure 9. Extracting class *Session* from *CashierRecord* during prefactoring

B.5. Actualization

Actualization expands on the explicit concept by adding new code which represents the full concept. Several new fields are added: `Calendar` `logoutTime`, `double sessionCash`, and `int transactions`.

B.6. Change Propagation

After the actualization of the feature, the change was propagated through the system, which is shown in figure 10. For example, the *Cashier* class was changed because the methods `login()` and `logout()` were updated to create a new session when logging in, and then to store the session data when logging out.

B.7. Postfactoring

At this point, class *CashierRecord* acts as a middle-man between *Cashier* and *Session*. This means that *Cashier* calls methods in *CashierRecord* that delegate

responsibility to the *Session* class. At this point, there are only two methods delegating responsibility in this way; it is determined that it is better to hide the functionality of *Session* from *Cashier* through delegation.

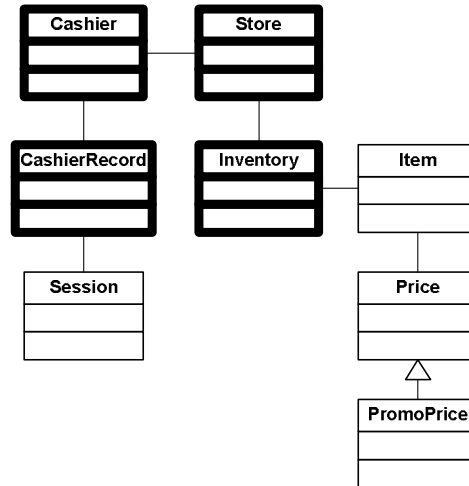


Figure 10. Classes visited during change propagation

B.8. New Baseline

Again, after the IC is completed, the code is committed into the version control system and is considered the new baseline that future IC's will be built onto. At this point, the user story is implemented in full and can be reviewed by the customer. This will ensure that the feature is implemented exactly as the customer was envisioning.

B.9. Testing

Class *Session* was created using the extract class refactoring on fields and methods in the *CashierRecord* class. Because of this, two assertions associated with methods moved from *CashierRecord* to *Session* are moved to *Session*'s test class. We also created 10 new assertions.