Abstract

We present a model of incremental change (IC) 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.

We argue that CID complements those agile processes that concentrate on team and management issues, and therefore can be combined with them.
Introduction

- Incremental Change (IC) is the process of adding new functionality to existing code
  - Foundation of software maintenance, evolution, and agile processes
- IC is less researched than refactoring
- Our aim is to fill this gap

IC process model

- Result of experience gained in previous case studies
- Activities of IC
  - Interactions with the world
  - IC design
  - IC Implementation

Initiation

- IC starts by a change request
- Change requests can be in the form of:
  - User stories
  - Change requests
  - Requirements documents
  - Bug report
- Prioritization of change requests, etc.
IC Design

Concept Location

- Concepts are extracted from change request
- Extracted concepts are located in the code and used as a starting point of IC
- Concepts can be implicit or explicit

Explicit concepts

- Implemented in software in a primitive form
  - IC will add new properties
- Example
  - Implement credit card payment
    - The current version implements only cash payment, represented as a double
Implicit Concepts

- Implied by the code but not expressed
  - IC will add them
  - Proper place still needs to be found
- Example
  - Implement authorization to open documents in word processor
    - In the current version, all users are authorized to open all documents

Impact Analysis

- Determine strategy and impact of change
- Classes identified in concept location make up the initial impact set
- Class dependencies are analyzed, and impacted classes are added to the impact set
  - Dependencies are explicit of hidden

Prefactoring

- Opportunistic refactoring that localizes (minimizes) impact of IC on software
- Extract Class (Fowler)
  - Gather fields, methods, and code snippets into a new class
- Extract Superclass
  - Create new abstract class
Actualization

- Creates new code
- Explicit concept: coding is done in the extracted class
- Implicit concept: create a new class(es)
  - “Plug-in” the new class by creating its instance in the old code

Change Propagation

- Beginning with the new class, visit neighboring classes and update them
- Change propagation is caused by
  - changes in public interfaces
  - changes in the semantics
    - pre/postcondition

Postfactoring

- Eliminate any anti-patterns that may have been introduced by IC
  - Long method
    - after added functionality, some methods may be doing too much
  - Man-in-the-middle
    - after repeated extraction, some classes may be doing only delegation
Testing

- Functional
  - Abott for GUI
- Unit
  - JUnit for Java
- Suite of tests has to be updated

New Baseline

- Check finished code into version control system
- New version serves as the new baseline for the next IC
- Use knowledge gained during IC to update software documentation

Concept-based Incremental Development (CID)

- Agile software development that uses repeated IC
- Software begins as a very simple initial implementation
- Functionality is added one step at a time
Case Study Goals

- Demonstrate CID
- Study the differences between outcome of
  - CID
  - Object-Oriented Analysis (Coad)
- Point of Sale
  - Initial implementation + 11 IC's

Sequence of CID steps

1. Initial version: single item sold, only cash payment, single price, etc.
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.
7. Add cashier session.

Sequence of IC's (cont.)

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.
Example: Add Cashier Session

- CID step 7
  - Explicit concept: “Session”
- Each login will start a new session
- Session data
  - Login/logout times
  - Number of transactions
  - Cash totals

Before IC

Concept Location

- Static dependency search begins at class containing `main()`
Impact Analysis

- Highlighted classes represent the impact set

Prefactoring

- Extract class Session from CashierRecord

Actualization

- Add new fields
  - logout time
  - total cash
  - total number of transactions
- Create methods to increment cash and transaction totals
Change Propagation

- **CashierRecord**
  - Changed to keep collection of sessions

- **Cashier**
  - Supporting methods were added

- **Store**
  - `commitSale()` method changed to update current session data

- **Inventory**
  - Visited, not changed

Postfactoring

- **CashierRecord** is a middle man
  - Eliminate or keep it?
    - Keeping it hides class details
    - Removing it makes maintenance easier
  - We kept it
    - only a couple of methods are delegated

Testing

- 7 test classes before IC
  - 55 assertions

- 8 test classes after IC
  - 1 new test class to test `Session (SessionTest)`
  - Relevant test methods moved from `CashierRecordTest` to `SessionTest`
  - 65 total assertions after IC
After IC

Case Study Testing Overview

<table>
<thead>
<tr>
<th>Step</th>
<th>Test Classes</th>
<th>Assertions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>34</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>40</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>53</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>55</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>68</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>71</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>77</td>
</tr>
<tr>
<td>11</td>
<td>13</td>
<td>89</td>
</tr>
<tr>
<td>12</td>
<td>14</td>
<td>98</td>
</tr>
</tbody>
</table>

Resulting OOD vs. CID architecture
Observations OOD vs. CID

- The core classes are remarkably similar
  - differences reflect the different philosophies
- OOD tries to anticipate future change
  - encapsulates some concepts for future evolution (Coad)
    - UPC, TaxCategory, ...
- CID encapsulates more concepts that were a topic in the past ICs
  - Inventory, CashierRecord

Related Work

  - Early proposed model of iterative software evolution
  - Incremental Change used to implement an unanticipated software feature
  - Continuation of the case study by Rajlich and Gosavi
  - Effects of Incremental Change on test suite
  - A tool that supports select activities of Incremental Change
Future Work

- Better refactoring tools for *extract class* and *remove middle man*
- Case studies where IC is combined with other agile methods that concentrate on organizational aspects
  - Scrum
- More detailed description of CID
  - A book?
Example Add Cashier Session

Concept location

```java
public class CashierRecord {

    private int cashierId;
    private String password;
    private String firstName;
    private String lastName;
    private boolean isLoggedIn;

    private Calendar loginTime;
    private Calendar lastLogin;

    public boolean isLoggedIn() {
        return isLoggedIn;
    }

    public String getName() {
        String fullName = firstName + " " + lastName;
        return fullName;
    }

    public Calendar getLoginTime() {
        return loginTime;
    }

    public void setLoginTime(Calendar c) {
        loginTime = c;
    }

    public String getLoginTimeString() {
        Date d = loginTime.getTime();
        SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yy hh:mm a");
        return sdf.format(d);
    }

    public void initLogin() {
        isLoggedIn = true;
        loginTime = Calendar.getInstance();
    }

    public void initLogout() {
        isLoggedIn = false;
        lastLogin = loginTime;
    }
}
```

Concept located here. The field `loginTime` will be used as a focal point for class extraction.

These methods will be extracted as well.

© Václav Rajlich, Neal Febbraro, 2007

14
Extracted class

```java
public class Session {

    private Calendar loginTime;

    public Session() {
        loginTime = new GregorianCalendar();
    }

    public Calendar getLoginTime() {
        return loginTime;
    }

    public void setLoginTime(Calendar c) {
        loginTime = c;
    }

    public String getLoginTimeString() {
        Date d = loginTime.getTime();
        SimpleDateFormat sdf =
            new SimpleDateFormat("MM/dd/yy hh:mm a");
        return sdf.format(d);
    }
}
```

Additional Session functionality will be coded in this new class created through the class extraction refactoring.
public class CashierRecordTest {

    CashierRecord testCR;
    Calendar today;

    @Before
    public void setUp() throws Exception {
        testCR = new CashierRecord(1234, "password", "John", "Doe");
    }

    @After
    public void tearDown() throws Exception {
        testCR = null;
    }

    @Test
    public void testGetName() {
        assertTrue(testCR.getName().equals("John Doe"));
    }

    @Test
    public void testGetLoginTimeString() {
        testCR.setLoginTime(new GregorianCalendar(2007, 6, 10, 12, 58));
        assertEquals("07/10/07 12:58 PM", testCR.getLoginTimeString());
    }
}

Methods extracted from the CashierRecord class will also have their associated unit tests moved to a new test class.
Unit Tests Extracted for Class Session

```java
public class SessionTest {

    private Session testSession;

    @Before
    public void setUp() throws Exception {
        testSession = new Session();
    }

    @After
    public void tearDown() throws Exception {
        testSession = null;
    }

    @Test
    public void testGetLoginTimeString() {
        testSession.setLoginTime(    
            new GregorianCalendar(2007,6,10,12,58));
        assertEquals("07/10/07 12:58 PM", testSession.getLoginTimeString());
    }
}
```

Additional test cases will be created by the programmer when new Session functionality is coded in the new class.