Bug-inducing analysis to prevent fault prone bug fixes

Yang Feng

Nanjing University
Introduction

- Empirical Study
- Focus on analyzing what is the most dangerous behavior in modifying code
- Focus on the Object-Oriented Programming
- Improve the SZZ tool
Bug-inducing analysis

Step 1: identify bug-fix changes (basis)

examine change log messages in two ways: searching for keywords such as "Fixed" or "Bug" and searching for references to bug reports like "#42233"
an explicitly recorded linkage between a bug tracking system and a specific SCM commit
Issue113
Desc: Support for AND/OR filter
Type: Enhancement

Issue111
Desc: Patch for
/trunk/slim3demo/src/slim3/demo/controller/checkbox/IndexController.java
Type: Patch

Issue106
Desc: DateUtil#toString() and locale
Type: Enhancement

Issue105
Desc: XG transactions do not work in dev server environment and unit testing
Type: Defect

Issue101
Desc: RequestHandler overwrite the parameters to attributes.
Type: Defect
Bug-inducing analysis

Step 2: trace backward to get bug-inducing changes

1. SZZ algorithm

2. Improvement of SZZ algorithm (we use)
SZZ algorithm

1. SZZ first finds bug-fix changes by locating bug identifiers or relevant keywords in change log text (finished in Step1)
SZZ algorithm

2. Run a diff tool to determine what changed in the bug-fixes
SZZ algorithm

Easy in code.google (in experiment we use DiffJ)
SZZ algorithm

Each different region is called a **hunk**

```java
/**
 * Constructor.
 *
 * @param attributeMeta the meta data of attribute
 * @throws NullPointerException
 * if the attributeMeta parameter is null
 */
public AbstractCriterion(AbstractAttributeMeta<?, ?> attributeMeta) {
    if (attributeMeta == null) {
        throw new NullPointerException(
            "The attributeMeta parameter must not be null.");
    }
    this.attributeMeta = attributeMeta;
}

/**
 * Compares its two arguments for order. Returns a negative integer, zero,
 * or a positive integer as the first argument is less than, equal to, or
 * greater than the second.
 */
```
SZZ algorithm

SZZ assumes that deleted or modified source code in each hunk is the location of a bug.
3. Tracks down the origins of deleted or modified source code using built-in \textit{annotate} feature of SCM systems (the annotate info only contains triples of \textit{current reversion line\#}, most recent modification revision, developer who made modification)
SZZ algorithm

Issue 113: Support for AND/OR filter

Affected files:
- Delete /trunk/slim3/src/main/java/org/slim3/datastore/Filter.java
- Modify /trunk/slim3/src/main/java/org/slim3/datastore/ModelQuery.java
- Delete /trunk/slim3/src/test/java/org/slim3/datastore/FilterTest.java
- Modify /trunk/slim3/src/test/java/org/slim3/datastore/ModelQueryTest.java

To get annotated hit filename link.
SZZ algorithm


/*
 * Copyright 2004-2010 the Season Foundation and the Others.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 * http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language
 * governing permissions and limitations under the License.
 */
package org.slim3.datastore;
import java.util.ArrayList;
import java.util.List;
import com.google.appengine.api.datastore.Text;
/**
 * An abstract class for criterion.
 * @author higa
 * @since 1.0.0
 */
public abstract class AbstractCriterion { 
/**
SZZ algorithm

It shows that the most recent modification is r1357, which SZZ considers it as bug-inducing change.
SZZ algorithm

We run a tool to find the differences between the bug-inducing commit (r1356-r1357) in the same method. And the tool DiffJ will give us the change types.
SZZ algorithm

For all modified files in bug-fix revision, do the same process above, get all the bug-inducing position. And include the change as a certain kind of change.
SZZ algorithm

However, SZZ is imprecise

1. view formatting change as bug-inducing change...

2. Not all the hunks are bug-fixes (blank lines, comments, formatting)
Improvement of SZZ algorithm

1. Use annotation graphs to provide more detailed annotation information

2. Ignore comment and blank line changes

3. Ignore format changes

4. Ignore outlier bug-fix revisions in which too many files were changed

5. Manually verify all hunks in the bug-fix changes
Improvement of SZZ algorithm

1. Use annotation graphs to provide more detailed annotation information (the recursive version of annotation feature)
Improvement of SZZ algorithm

2. Ignore comment and blank line changes

```java
public void notifySourceElementRequestor()
{
    -
    +    if (reportReferenceInfo) {
    +        notifyAllUnknownReferences();
    +    }
    // collect the top level ast nodes
    int length = 0;
```
Improvement of SZZ algorithm

3. Ignore format changes

```
- if ( folder == null ) return;
+ if (folder == null)
+    return;
```
Improvement of SZZ algorithm

4. Ignore outlier bug-fix revisions in which too many files were changed

Too many changed files exist in bug-fix change? It may be imprecise.
Bug-inducing analysis

Step 3: Transform bug-inducing change into a set of atomic changes

Their granularity matches our analysis, every atomic change has its own category,
Category of atomic changes

<table>
<thead>
<tr>
<th>Method Added</th>
<th>Parameter Name Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>codeChanged</td>
<td>parameterReordered</td>
</tr>
<tr>
<td>codeAdded</td>
<td>methodBlockAdded</td>
</tr>
<tr>
<td>typeDeclarationAdded</td>
<td>accessAdded</td>
</tr>
<tr>
<td>innerClassAdded</td>
<td>modifierChanged</td>
</tr>
<tr>
<td>fieldAdded</td>
<td>lookupChanged</td>
</tr>
<tr>
<td>importAdded</td>
<td>instanceFieldInitializerChanged</td>
</tr>
<tr>
<td>parameterAdded</td>
<td>staticFieldInitializerChanged</td>
</tr>
<tr>
<td>returnTypeChanged</td>
<td>instanceInitializerAdded</td>
</tr>
<tr>
<td>accessChanged</td>
<td>emptyInstanceInitializerDeleted</td>
</tr>
<tr>
<td>constructorAdded</td>
<td>instanceInitializerChanged</td>
</tr>
<tr>
<td>throwsAdded</td>
<td>staticInitializerAdded</td>
</tr>
<tr>
<td>parameterTypeChanged</td>
<td>staticInitializerDeleted</td>
</tr>
<tr>
<td>variableChanged</td>
<td>staticInitializerChanged</td>
</tr>
</tbody>
</table>

These types are concluded from the tool DiffJ and related previous paper.

So some of the atomic changes are checked by the tool, and some of them are checked manually.
Bug-inducing analysis
Step 4: count category of atomic change about every bug-inducing change
Bug-inducing analysis

Step 5: combing all statistics about every bug-inducing change
In our experiment, we investigated three projects Jedit, protostuff, encog respectively. And we drew the same conclusion in some aspect.
problem

We find that the type *codeAdded* and *codeChanged* are more dangerous than other types in all three projects.

So we do further investigation in the two change types.
We could not just draw conclusion through `codeAdded` or `codeChanged`. So we check all `codeAdded` and `codeChanged` changes and classify them in detail.
It shows that if/else clause changes in *codeAdded* or *codeChanged* are more dangerous.
Another problem

we find that typeDeclarationAdded would cause less bugs in all projects. (typeDeclarationAdded Means add a class in fact)
How to avoid danger?

1. apply widely recognized software design patterns and strict object-oriented rules
2. Use Open/Closed Principle to build software.
Future work

1. A much wider selection of projects

2. with the number of projects grown, Other change types except for what we have discussed above may also reveal some regular patterns
Questions?