Empirical Investigation of Code Comments Correlation

Snigdha Brahma (u5612813)
Supervised By: Dr. Zhenchang Xing
Research School of Computer Science
Australian National University
Overview

• Introduction
• Objectives
• Motivation
• Review of Literature
• Approach
• Detailed Approach
• Result
• Conclusion & Future work
Introduction

“Good software development practice requires that a sufficient number of comments will include in computer software in order to adequately document the operation of the software”[1]

“The ability to use a computer to analyze existing computer software and automatically determine its characteristics has been a long term computer science research goal”[1]

Objectives

- To find the important/key methods and their comments
- To observe the correlation between the code and the comments.
- To use sub language to describe the comments.
Motivation

• Define formally a comment structure. (How it looks like)
• Find how comments are important to key methods.
• Check which part of the comment is important.
• Define certain aspects of the comment structure.
Review of Literature

Approach

- Reverse Engineering

- Extract Information

- Calculate and Analyze extraction result

- Result (to support motivation)
Detailed approach

- Source: Java source code.
- Parse java source code with Eclipse AST Parser
- Java source file represented as tree of AST nodes.
- Nodes: sub – class of ASTNode.
- Nodes (Specialized for an element of Java Programming Language):
  - method declarations
  - variable declaration
  - assignments
  - method invocations
  - comments and so on
Detailed approach

- Information extracted:
  - Method declarations
  - Invoked Methods
  - Class information
  - Package information

- To determine the key method, we considered 6 situations:
  - Method called by another method in the same class (situation 1)
  - Method called by another method in a different class (situation 2)
  - Method called by another method in different package (situation 3)
  - Method calling another method in the same class (situation 4)
  - Method calling another method in a different class (situation 5)
  - Method calling another method in a different package (situation 6)
Detailed Approach

- Random weights were assigned to each situation. For example:
  - Method called by another method in the same class (higher weight)
  - Method called by another method in a different class (comparatively lower weight)
  - Method called by another method in different package (very low weight)
  - Method calling another method in the same class (higher weight)
  - Method calling another method in a different class (comparatively lower weight)
  - Method calling another method in a different package (very low weight)
Detailed Approach

• Key method:
Consider,
Ratio
\[
\frac{(x_1w_1 + x_2w_2 + x_3w_3 + 1)}{(x_4w_4 + x_5w_5 + x_6w_6 + 1)}
\]
Where,
\(x_1 = \) count of situation 1; \(x_2 = \) count of situation 2;
\(x_3 = \) count of situation 3; \(x_4 = \) count of situation 4;
\(x_5 = \) count of situation 5; \(x_6 = \) count of situation 6;
w_1, w_2, w_3, w_4, w_5, w_6 are weights allotted for each situations.
Detailed Approach

- Comments of the key methods were observed based on the given set of comment aspects.

Fig1: Comment Aspects
<table>
<thead>
<tr>
<th>Package name</th>
<th>Method name</th>
<th>Method Path</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.compiler.dao.LoginBean</td>
<td>forward</td>
<td>3.436</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.StartAccount</td>
<td>getUsername</td>
<td>1.204</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.LoginBean</td>
<td>forward</td>
<td>15.517</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleDirectionHibernateDAO</td>
<td>deleteDir</td>
<td>2.263</td>
<td></td>
</tr>
<tr>
<td>com.compiler.beans.mb.login.StartAccount</td>
<td>getUsername</td>
<td>0.063</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleAccountHibernateDAO</td>
<td>deleteUser</td>
<td>1.995</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleDirectionHibernateDAO</td>
<td>updateDir</td>
<td>2.263</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.LoginBean</td>
<td>login</td>
<td>1.956</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.LoginBean</td>
<td>login</td>
<td>9.058</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleAccountHibernateDAO</td>
<td>updateGenericAccountInfo</td>
<td>1.250</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.LoginBean</td>
<td>updateDir</td>
<td>0.874</td>
<td></td>
</tr>
<tr>
<td>com.compiler.beans.mb.login.LoginBean</td>
<td>forward</td>
<td>1.769</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleAccountHibernateDAO</td>
<td>getUser</td>
<td>8.312</td>
<td></td>
</tr>
<tr>
<td>com.compiler.beans.mb.login.LoginBean</td>
<td>login</td>
<td>1.027</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.StartAccount</td>
<td>getUsername</td>
<td>0.086</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleProjectHibernateDAO</td>
<td>deleteProject</td>
<td>2.588</td>
<td></td>
</tr>
<tr>
<td>com.compiler.dao.hibernate.OracleProjectHibernateDAO</td>
<td>updateGenericProjectInfo</td>
<td>1.602</td>
<td></td>
</tr>
</tbody>
</table>
Results (Contd.)

- Method called by another method in the same class, $w_1 = 0.5$
- Method called by another method in a different class, $w_2 = 0.3$
- Method called by another method in different package, $w_3 = 0.07$
- Method calling another method in the same class, $w_4 = 0.5$
- Method calling another method in a different class, $w_5 = 0.2$
- Method calling another method in a different package, $w_6 = 0.05$
Results (Contd.)

- Methods with higher ratio values:
  - com.compiler.dao.hibernate.LoginBean.forward : 15.517
  - com.compiler.dao.hibernate.LoginBean.login : 9.058
  - com.compiler.dao.hibernate.OracleAccountHibernateDAO.getUser : 8.312
Results (Contd.)

Comment information of the methods with high ratio value:

No comment information available for the method OracleAccountHibernateDAO.getUser

```java
public GenericAccountInfo getUser(String login) {
    Session session = factory.openSession();
    Transaction tx = null;
    List<GenericAccountInfo> accountsInfo = new ArrayList<GenericAccountInfo>();
    try {
        tx = session.beginTransaction();
        accountsInfo = session.createQuery("from GenericAccountInfo gai where gai.email = :login ").setParameter("login", login).list();
        tx.commit();
    } catch (HibernateException e) {
        if (tx != null)
            tx.rollback();
        e.printStackTrace();
    } finally {
        session.close();
    }
    return accountsInfo.isEmpty() ? null : accountsInfo.get(0);
}```
Results (Contd.)

Source Project Path:
C:\Users\Snigdha\Desktop\testJavaSourceCode
Method Name : Ratio
org.eclipse.jdt.core.dom.AST.convertCompilationUnit : 1.000
/**
 * Internal method.
 * <p>
 * This method converts the given internal compiler AST for the given source string
 * into a compilation unit. This method is not intended to be called by clients.
 * </p>
 *
 * @param level the API level; one of the <code>JLS*</code> level constants
 * @param compilationUnitDeclaration an internal AST node for a compilation unit declaration
 * @param options compiler options
 * @param workingCopy the working copy that the AST is created from
 * @param monitor the progress monitor used to report progress and request cancellation,
 * or <code>null</code> if none
 * @param isResolved whether the given compilation unit declaration is resolved
 * @return the compilation unit node
 * @since 3.4
 * @noreference This method is not intended to be referenced by clients.
 */

public static CompilationUnit convertCompilationUnit(
    int level,
    org.eclipse.jdt.internal.compiler.ast.CompilationUnitDeclaration compilationUnitDeclaration,
    Map options,
    boolean isResolved,
    org.eclipse.jdt.internal.core.CompilationUnit workingCopy,
    int reconcileFlags,
    IProgressMonitor monitor) {

    ASTConverter converter = new ASTConverter(options, isResolved, monitor);
    AST ast = AST.newAST(level);
    int savedDefaultNodeFlag = ast.getDefaultNodeFlag();
    ast.setDefaultNodeFlag(ASTNode.ORIGINAL);
    BindingResolver resolver = null;
    if (isResolved) {
        resolver = new DefaultBindingResolver(compilationUnitDeclaration.scope, workingCopy.owner, new DefaultBindingResolver.BindingTables(), false, true);
        ((DefaultBindingResolver) resolver).isRecoveringBindings = (reconcileFlags & ICCompilationUnit.KNARX_BINDINGS_RECOVERY) != 0;
        ast.setFlag(AST.RESOLVED_BINDINGS);
    } else {
        resolver = new BindingResolver();
    }
    ast.setBindingResolver(resolver);
    converter.setAST(ast);

    CompilationUnit unit = converter.convert(compilationUnitDeclaration, workingCopy.getSource());
    unit.setLineEndTable(compilationUnitDeclaration, compilationUnitDeclaration.getCompilationResult().getSource().getLineSeparatorPositions());
    unit.set腴Template(workingCopy.originalFromClone());
    ast.setDefaultNodeFlag(savedDefaultNodeFlag);
    return unit;
}
The comment of the method has the following information:

- Gives a brief introduction on what the method does.
- Version is mentioned.
- Parameters were introduced with its role and purpose.
- Describes method behavior.
Conclusion & Future Work

- Get a threshold value to judge the key method.
- Use some natural language process technique to deal with comments (use NLP/ML technique).
- Introduce other technique to come up with UI for the comment maintenance.
Questions