A Human-Centric Approach to Program Understanding

“The real question is not whether machines think, but whether men do.”

-- B. F. Skinner

Readability  Runtime Behavior  Documentation

Ray Buse - PhD Proposal
University of Virginia, Department of Computer Science
1.20.2010
Maintenance accounts for about **70-90%** of the total lifecycle budget of a software project.\(^1,2\)


Reading Code is the most time consuming part of all maintenance activities.$^{3,4,5}$


“Understanding code is by far the activity at which professional developers spend most of their time.”

Reading Code is the most Poorly Understood Software Engineering activity.\textsuperscript{7,8}


Reading Code is the most **Poorly Understood** Software Engineering activity.
Understanding is difficult to...

**Model**
- Based on a complex combination of factors

**Evaluate**
- Lack of established metrics/baselines
- User studies are unattractive
Two Key Insights

**Machine Learning** allows us to combine many semantically shallow features of code to gain new deep insights.

**PL Techniques** can be adapted to generate documentation artifacts that are directly comparable to human created ones.
Thesis

We can combine insights from **Machine Learning** and **Programming Languages** to

- **Model** aspects of code understanding *accurately* and
- **Generate** output that compares favorably with human documentation.
Proposal: Three Dimensions of Understanding

- Readability
- Runtime Behavior
- Documentation
Proposal: Three Dimensions of Understanding

- **Readability**: Textual characteristics that make code understandable.
- **Runtime Behavior**: Structural characteristics that help developers understand what a program is expected to do.
- **Documentation**: Non-code text that helps developers understand a program.
Research Projects

Metrics for:

• Code Readability
• Path Execution Frequency

Algorithms for Documentation of:

• Exceptions
• Code Changes
• APIs
Broader Impact

New algorithms and metrics to support:

• Software Development and Composition
  – Metrics for Software Quality Assurance
  – Automatic Documentation

• Software Analysis
  – Runtime Behavior model for optimizing compilers
  – Metrics for targeting analyses, prioritizing output, and evaluating research
The rest of this proposal

• A review of each proposed contribution
  – Technical Merit
  – Evaluation Strategy
  – Related Work

• Research timeline and other bookkeeping

• Concluding Remarks
Metrics for:
- **Code Readability**  
  ISSTA ‘08  
  TSE ‘10
- **Path Execution Frequency**  
  ICSE ‘09

Algorithms for Documentation of:
- **Exceptions**  
  ISSTA ‘08
- **Code Changes**
- **APIs**  
  Published  
  In Progress
Metrics for:

• **Code Readability**  

• **Path Execution Frequency**

Algorithms for Documentation of:

• **Exceptions**

• **Code Changes**

• **APIs**

---

ISSTA ‘08  TSE ‘10

ICSE ‘09

Published

In Progress
/**
 * Extend this Execution path by one level.
 * @throws IllegalStateException If the move path invalid..
 */
private List<ExecutionPath> extend (ExecutionPath ep)
{
    paths = new LinkedList<ExecutionPath>();
    Unit last = ep.getLast();
    List<Unit> succs = graph.getSuccsOf(last);
    //this is the end of the path
    if (succs.isEmpty())
    {
        ep.setComplete(true);
        paths.add(ep);
        return paths;
    }
    if (succs.size() == 1)
    {
        Unit s = succs.get(0);
        if (ep.contains(s))
        {
            //do nothing
        }
        else
        {
            ep.addLast(s);
            if (graph.getTails().contains(s))
            {
                ep.setComplete(true);
            }
        }
    }
}
Hypothesis

With a simple set of textual features, we can derive from a set of human judgments an accurate model of readability for code.

Success depends on

- Gathering human judgments
- Choosing predictive textual features
Data Gathering

• We asked 120 students at UVa to rate the readability of a set of snippets...
Data Set

Vertical bands indicate snippets were agreement was high
Choosing predictive textual features

We choose local code features

- Line length
- Length of identifier names
- Comment density
- Blank lines
- Presence of numbers
- [and 20 others]

Modeled with a Bayesian Classifier
Model Performance

Model agrees with humans as much as they agree with each other on average

Spearman correlation between annotator scores and average scores

Human Annotators (sorted)

our metric

average human
Related Work

• Readability metrics for natural languages
  – Very popular, DOD standards etc

• In the software domain
  – Complexity metrics (often used, but utility is questionable)
Conclusions

We can automatically judge readability about as well as the average human can.

This notion of readability shows significant correlation with:

- Code churn
- A bug finder
- Program maturity
Metrics for:

- Code Readability
- Path Execution Frequency

Algorithms for Documentation of:

- Exceptions
- Code Changes
- APIs

Published

In Progress
/**
 * Extend this Execution path by one level.
 * @throws IllegalStateException If the move path invalid.
 */
private List<ExecutionPath> extend (ExecutionPath ep)
{
    paths = new LinkedList<ExecutionPath>();
    Unit last = ep.getLast();
    List<Unit> succs = graph.getSuccsOf(last);
    //this is the end of the path
    if (succs.isEmpty())
    {
        ep.setComplete(true);
        paths.add(ep);
        return paths;
    }
    if (succs.size() == 1)
    {
        Unit s = succs.get(0);
        if (ep.contains(s))
        {
            //do nothing
        }
        else
        {
            ep.addLast(s);
            if (graph.getTails().contains(s))
            {
                ep.setComplete(true);
            }
        }
    }
}
Key Idea

- Developers often have *expectations* about common and uncommon cases in programs.

- The *structure* of code they write can sometimes reveal these expectations.
public V put(K key, V value) {
    if (value == null) {
        throw new Exception();
    }
    if (count >= threshold) {
        rehash();
    }
    index = key.hashCode() % length;
    table[index] = new Entry(key, value);
    count++;
    return value;
}

*simplified from java.util.HashTable jdk6.0
Hypothesis

We can *accurately* predict the runtime frequency of program paths by analyzing their static *surface features*.

Goal:

• Know what programs are *likely* to do without having to run them (produce a static profile)
Applications for Static Profiles

Indicative (dynamic) profiles are often unavailable

Profile information can improve many analyses

- Profile guided optimization
- Complexity/Runtime estimation
- Anomaly detection
- Significance of difference between program versions
- Prioritizing output from other static analyses
Approach

**Model** path with a set of features that may correlate with runtime path frequency

**Learn** from programs for which we have indicative workloads, we used a Logistic Regression

**Predict** which paths are most or least likely in other programs
Evaluation

Choose 5% of all paths and get 50% of runtime behavior.

Ranking by our metric

Baseline: random ranking

Percent of all paths selected

Time spent on paths
**Evaluation**

Choose 1 path per method and get **94%** of runtime behavior.

Baseline: random ranking

Ranking by our metric

Time spent on paths vs. Number of paths selected per method.
Related Work

• Static Branch Prediction [Ball & Larus ’92]
  – For each branch, which direction is most likely
  – In a direct comparison, our tool is better
Conclusion

- A formal model that statically predicts relative dynamic path execution frequencies
- The promise of helping other program analyses and transformations
Metrics for:

- Code Readability
- Path Execution Frequency

Algorithms for Documentation of:

- Exceptions
- Code Changes
- APIs

Published

In Progress

ISSTA ‘08
TSE ‘10
ICSE ‘09
/**
 * Extend this Execution path by one level.
 * @throws IllegalStateException
 *   If the move path invalid.
 */

private List<ExecutionPath> extend (ExecutionPath ep) {
    paths = new LinkedList<ExecutionPath>();

    Unit last = ep.getLast();
    List<Unit> succs = graph.getSuccsOf(last);

    //this is the end of the path
    if (succs.isEmpty()) {
        ep.setComplete(true);
        paths.add(ep);
        return paths;
    }

    if (succs.size() == 1) {
        Unit s = succs.get(0);
        if (ep.contains(s)) {
            //do nothing
        } else {
            ep.addLast(s);
            if (graph.getTails().contains(s)) {
                ep.setComplete(true);
            }
        }
    }
}

Documentation
Generate for:
- Exceptions
- APIs
- Version Changes

Key: Use symbolic execution and summarization heuristics to generate human-readable results.
Use

• For Internal Developers
  – Easier to keep track of what’s going on

• For Maintenance and Testing
  – Easier to read old code.

• For External Developers
  – Easier to integrate off-the-shelf software libraries
Three Types of Documentation

- Exceptions
- Code Changes
- APIs
Documenting Exceptions

```java
/**
 * @throws Exception If the value is null
 */
public V put(K key, V value)
{
    if (value == null)
        throw new Exception();

    if (count >= threshold)
        rehash();

    index = key.hashCode() % length;
    ...
```

*best practice dictates that exceptions should be documented*
Documenting Exceptions

```java
/**
 * @throws Exception If the value is null
 */
public V put(K key, V value) {
    if (value == null)
        throw new Exception();
    if (count >= threshold)
        rehash();
    index = key.hashCode() % length;
    ...
```

Best practice dictates that exceptions should be documented.

Does this method throw an exception?

*simplified from java.util.HashTable jdk6.0*
Importance

Mishandling or Not handling can lead to:

• Security vulnerabilities
• May disclose sensitive implementation details
• Breaches of API encapsulation
• Any number of minor to serious system failures
Hypothesis

Mechanical documentation of exceptions can be *at least as good* as human on average.

- More complete
- More accurate

We extract paths to *throw* statements and use symbolic execution to *generate path predicates*
Examples

• Sometimes we do **better**:  

  Worse: id == null  
  (Us) Better: id is null or id.equals("""")

• Sometimes we do about the **same**:  

  Same: has an insufficient amount of gold.  
  (Us) Same: getPriceForBuilding() > getOwner().getGold()

• Sometimes we do **worse**:  

  Better: the queue is empty  
  (Us) Worse: private variable m_Head is null
Our documentation is as good as human over 80% of the time.
Code Change Examples

jfreechart rev 3405

(start): Changed from Date to long,
(end): Likewise,
(getStartMillis): New method,
(getEndMillis): Likewise,
(getStart): Returns new date instance,
(getEnd): Likewise.

Phex 3542

Minor change

Jabref rev 2917

Fixed NullPointerException when downloading external file and file directory is undefined.
Subject: An appeal for more descriptive commit messages

I know there is a lot going on but please can we be a bit more descriptive when committing changes. Recent log messages have included:
"some cleanup"
"more external service work"
"Fixed a bug in wiring"
which are a lot less informative than others...


Toby,

Going forward, could you I ask you to be more descriptive in your commit messages? Ideally you should state what you've changed and also why (unless it's obvious)... I know you're busy and this takes more time, but it will help anyone who looks through the log ...


Sorry to be a pain in the neck about this, but could we please use more descriptive commit messages? I do try to read the commit emails, but since the vast majority of comments are "CAY-XYZ", I can't really tell what's going on unless I then look it up.

Key Idea

Generate Documentation that describes the **effect** of a change on the runtime behavior of a program

– What conditions are necessary to activate the change

– What the new behavior is
Algorithm

- Generate predicates for each statement
- Compare predicates across versions
- Summarize change and distill structured output

When \( X \),
- Do \( Y \)
- Instead of \( Z \)
Evaluation

Our documentation is as good as human over 80% of the time
API Usage Documentation

“The greatest obstacle to learning an API ... is insufficient or inadequate examples” ⁹

Java Util ObjectOutputStream

FileOutputStream fos = new FileOutputStream("t.tmp");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(new Date());
oos.close();

Java Util BufferedReader

BufferedReader in = new BufferedReader(new FileReader("foo.in"));

Weka Core Instance

// Create the instance
Instance iExample = new Instance(4);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(0), 1.0);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(1), 0.5);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(2), "gray");
iExample.setValue((Attribute)fvWekaAttributes.elementAt(3), "positive");
isTrainingSet.add(iExample);
Key Idea

Combine insights from **specification mining**, automatic documentation, and code summarization.

Specification mining **false positives** – usage patterns that are common but aren't required – are exactly what we want to find.
Algorithm

Given a target class to document, and a set of code files that use the class (e.g., mined from the web).

– **Model** usages of the classes as a finite state machine or regular expression
– **Combine** machines that are similar
– **Output** most common machines as usage examples
Evaluation

Manual comparison to JavaDoc examples

• Are we able to come up with the same examples?
  – Precision / Recall / F-measure

• User Study
Conclusion

To create algorithms for three types of documentation:

– Exceptions
– Code Changes
– API Usage

Evaluate by comparing to human generated documentation and/or with a user study
Research Timeline

- Original Readability Metric
- Documentation for Exceptions
- Runtime Path Frequency
- Journal Readability Metric
- Documentation for Changes
- Documentation for APIs

- Research Period
- Publishing Lag

- 2007
- 2008
- 2009
- 2010
- 2011

- today

- expected graduation
A 2005 NASA survey found that the most significant barrier to code reuse is that software is “too difficult to understand” or is “poorly documented.”

Conclusion: Understanding programs at many levels

- How easy is it to understand and maintain this software? Readability

- Where are the corner cases, and where are the common paths? Runtime Behavior

- How can this code go wrong? Documenting Exceptions

- How do I use this code? Documenting APIs

- What does proposed fix really do? Documenting Changes
All Questions Encouraged

These slides, the proposal document, and much more information is available at:

http://arrestedcomputing.com/proposal

Readability  Runtime Behavior  Documentation

Thanks for Coming!