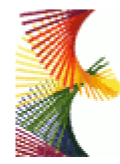
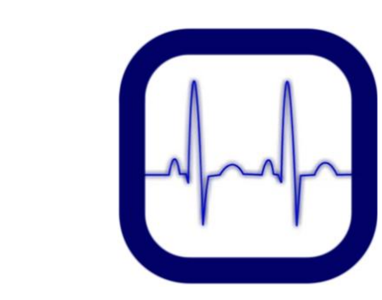

Helix

A Human-Centric Approach to Program Understanding

"The real question is not whether machines think,
but whether men do." -- B. F. Skinner



Readability

Documentation

Ray Buse - PhD Proposal
University of Virginia, Department of Computer Science

1.20.2010

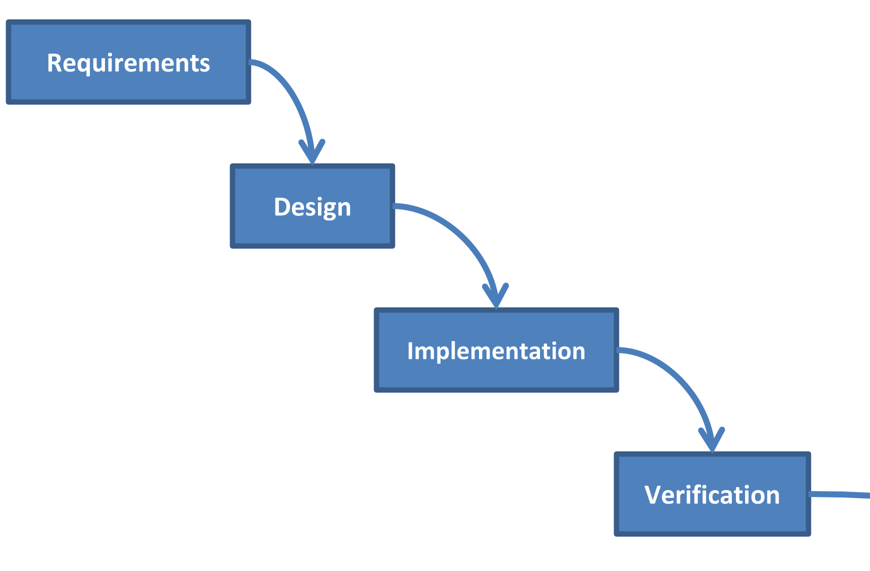
Requirements

Design

Implementation

Verification

Maintenance



Maintenance accounts for about

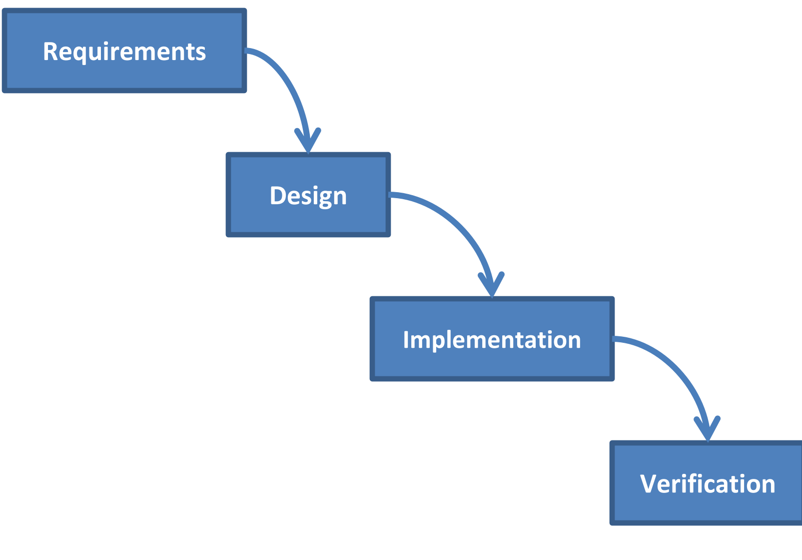
70-90%

of the total lifecycle budget of a software project.[1,2]

Maintenance

1. T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.

2. R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies*, Engineering Process and Business Practices. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.



Reading Code is the most time consuming part of all maintenance activities.[3,4,5]

Maintenance

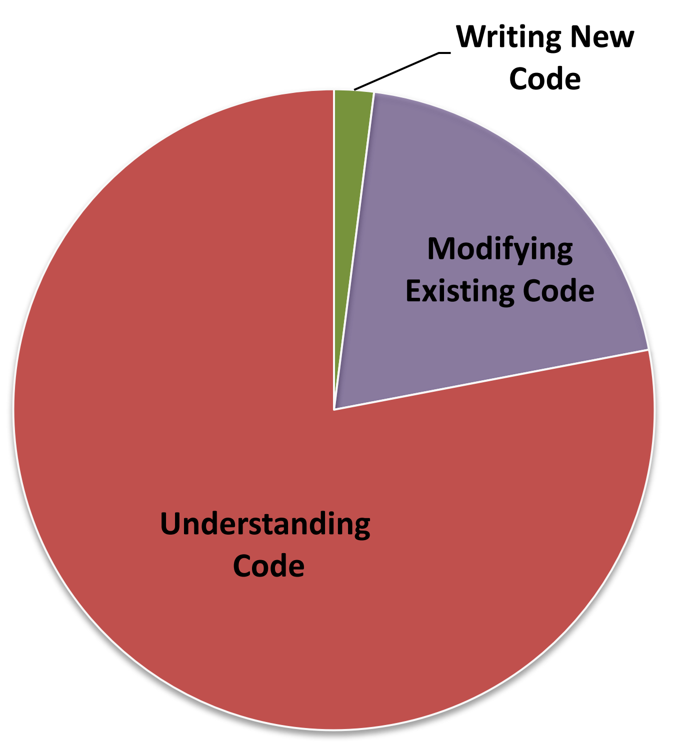
Reading Code

3. L. E. Deimel Jr. *The uses of program reading*. SIGCSE Bull., 17(2):5-14, 1985.

4. R. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley, 2003.

5. S. Rugaber. *The use of domain knowledge in program understanding*. Ann. Softw. Eng.,(1-4):143-192, 2000.

"**Understanding code** is **by far** the activity at which professional developers spend most of their time." [6]



6. Peter Hallam. *What Do Programmers Really Do Anyway?* Microsoft Developer Network (MSDN) – C# Compiler. Jan 2006.

Reading Code is the most
Poorly Understood
Software Engineering activity.[7,8]

7. D. Parnas. *Software aging*. In Software Fundamentals. Addison-Wesley, 2001.

8. D. Zokaities. *Writing understandable code*. In Software Development, pages 48-49, jan 2002.

Reading Code is the most
Poorly Understood
Software Engineering activity.

	all	program understanding
ICSE	4,387	16
PLDI	780	1

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 **understand**able.

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

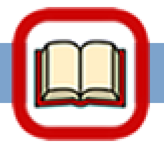
Broader Impact

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:

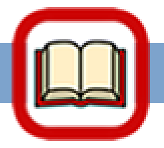
- Code Readability ISSTA '08 TSE '10
- Path Execution Frequency ICSE '09

Algorithms for Documentation of:

- Exceptions ISSTA '08
- Code Changes
- APIs

Published

In Progress



Metrics for:

- Code Readability ISSTA '08 TSE '10
- Path Execution Frequency ICSE '09

Algorithms for Documentation of:

- Exceptions ISSTA '08
- 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);
            }
```

Readability

Model human judgments about code readability

Create a readability **metric**

Key: Use textual features to *approximate* human judgments

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…

```
/**
 * Computes factorial with recursion
 */
public int factorial( int integer )
{
  if( integer < 1 )
    return 0;

  if( integer == 1)
    return 1;

    return integer * factorial( integer - 1 );
```

Snippet Pack demo: 2 of 4

1 2 3 4 5

Data Set

Vertical bands indicate snippets were agreement was high

120 human annotators

100 snippets

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

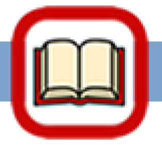
Model Performance

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 ISSTA '08 TSE '10
- Path Execution Frequency ICSE '09

Algorithms for Documentation of:

- Exceptions ISSTA '08
- Code Changes
- APIs

Published

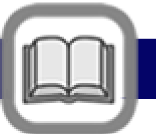
In Progress

```
/**
 * 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);
            }
```

Runtime Behavior

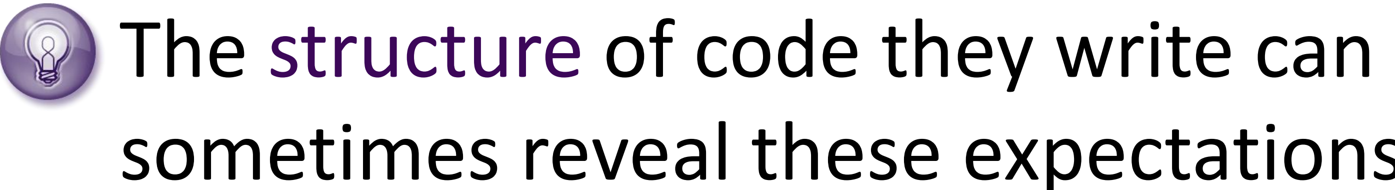
Model path execution frequency statically

Key: Use path surface features to uncover developer expectations

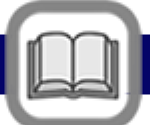


Key Idea

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



- The structure of code they write can sometimes reveal these expectations



Intuition

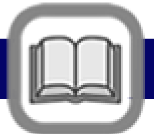
```
public V put(K key , V value)
{
    if ( value == null )
      throw new Exception();

    if ( count >= threshold )
      rehash();

    index = key.hashCode() % length;

    table[index] = new Entry(key, value);
    count++;

    return value;
}
```

Exception

Invocation that changes a lot of the object state

Some computation

*simplified from java.util.HashTable jdk6.0

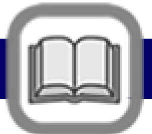


Hypothesis

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*)

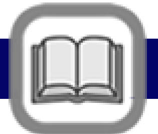


Applications for Static Profiles

Indicative (dynamic) profiles are often unavailable

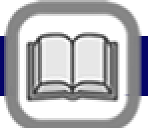
Profile information can improve many analyses

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

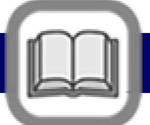


Approach

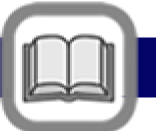
- **Model** path with a set of features that may correlate with runtime path frequency
- **Learn** from programs for which we have indicative workloads, we used a Logistic Regression
- **Predict** which paths are most or least likely in other programs



Evaluation

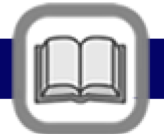


Evaluation



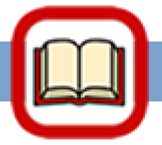
Related Work

- 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 ISSTA ‘08 TSE ‘10
- Path Execution Frequency ICSE ‘09

Algorithms for Documentation of:

- Exceptions ISSTA ‘08
- Code Changes
- APIs

Published

In Progress

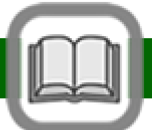
```
/**
 * Extend this Execution path by one level.
 *
 * @throws IllegalStateException If the move path invalid..
 */
private List<ExecutionPath> extend (ExecutionPath ep)
{
    paths = new LinkedList<ExecutionPath>();

    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);
            }
```

Documentation

Generate for:

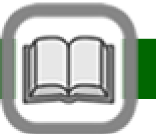
- Exceptions
- APIs
- Version Changes

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



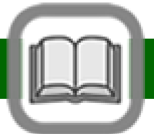
Use

- 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

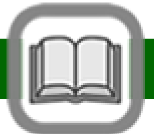


Documenting Exceptions

```
/**
* @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

*simplified from java.util.HashTable jdk6.0



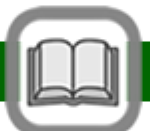
Documenting Exceptions

```
/**
* @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:

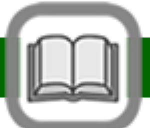
- Security vulnerabilities
- May disclose sensitive implementation details
- Breaches of API encapsulation
- Any number of minor to serious system failures

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**:

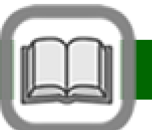
```
      Worse:  id == null
(Us) Better: id is null or id.equals("")
```

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

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

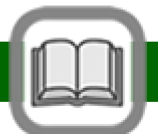
- Sometimes we do **worse**:

```
      Better: the queue is empty
(Us) Worse:  private variable m_Head is null
```



Key Results

Our documentation is as good as human over **80%** of the time



Code Change Examples

CVS

jfreechart rev 3405

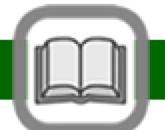
(start): Changed from Date to long,
(end): Likewise,
(getStartMillis): New method,
(getEndMillis): Likewise,
(getStart): Returns new date instance,
(getEnd): Likewise.

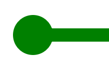
Phex 3542

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 commiting 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...

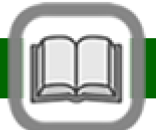
http://osdir.com/ml/apache.webservices.tuscany.devel/2006-02/msg00227.html

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 ...

http://lists.macosforge.org/pipermail/macports-dev/2009-June/008881.html

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.

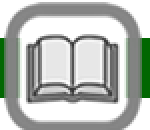
http://osdir.com/ml/java.cayenne.devel/2006-10/msg00044.html



Key Idea

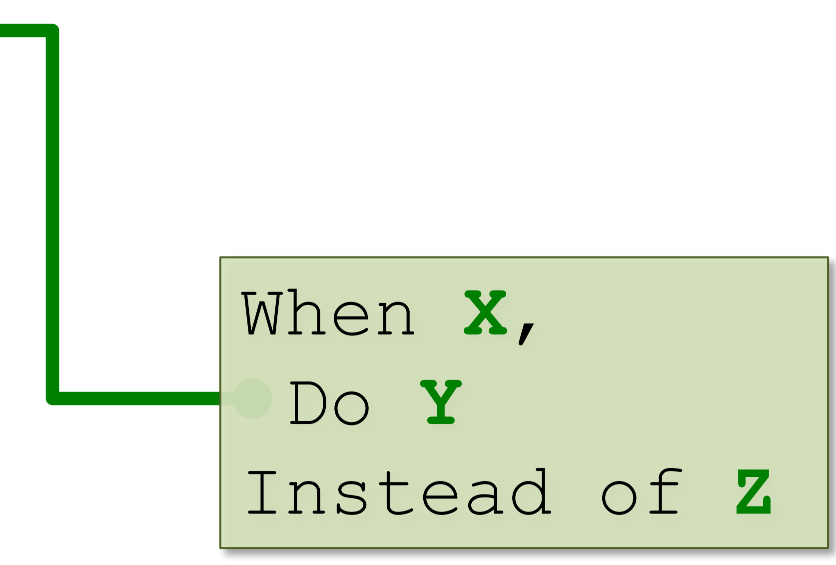
Generate Documentation that describes the effect of a change on the runtime behavior of a program

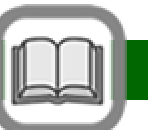
- What conditions are necessary to activate the change
- What the new behavior is



Algorithm

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





Our documentation is as good as human over **80%** of the time

Percent of log messages in sample

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

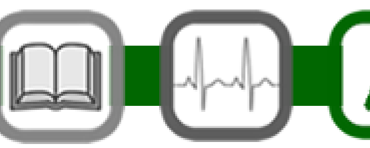
freecol
itext
jfreechart
jabref
phex
total

Worse
Same
Better

API Usage Documentation

"The greatest obstacle to learning an API … is insufficient or inadequate examples" [9]

9. M. P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. IEEE Softw., 26(6):27-34, 2009.



API Usage Documentation

java.util.ObjectOutputStream

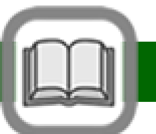
```
FileOutputStream fos = new
            FileOutputStream("t.tmp");
ObjectOutputStream oos = new
            ObjectOutputStream(fos);
oos.writeInt(12345);
oos.writeObject("Today");
oos.writeObject(
oos.close();
```

java.util.BufferedReader

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

weka.core.Instance

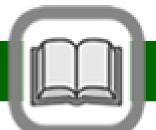
```
// Create the instance
Instance iExample = new Instance(4);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(0), 1.0);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(1), 0.5);
iExample.setValue((Attribute)fvWekaAttributes.elementAt(2), "gray");
iExample.setValue((Attribute)fvWekaAttributes.elementAt(3),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

 isTrainingSet.add(iExample);
```



Key Idea

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

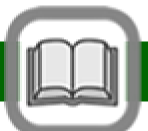
Specification mining *false positives* – usage patterns that are common but aren't required – are exactly what we want to find.



Algorithm

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

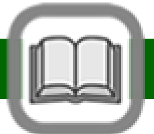
- Model usages of the classes as a finite state machine or regular expression
- Combine machines that are similar
- Output most common machines as usage examples



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:

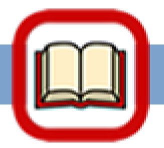
- Exceptions
- Code Changes
- API Usage

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

Research Timeline

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.” [10]

10. Nasa Software Reuse Working Group. Software reuse survey. http://www.esdswg.com/softwarereuse/Resources/library/working_ group_ documents/survey2005, 2005.



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!