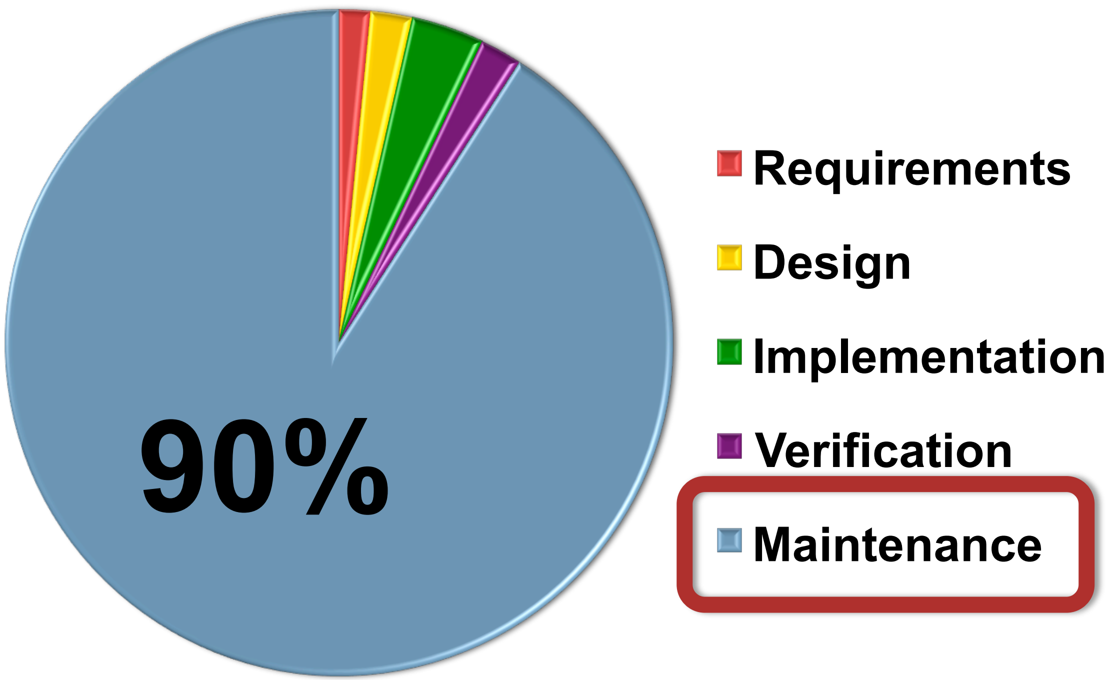
LEVERAGING LIGHTWEIGHT ANALYSES TO AID SOFTWARE MAINTENANCE

ZAK FRY

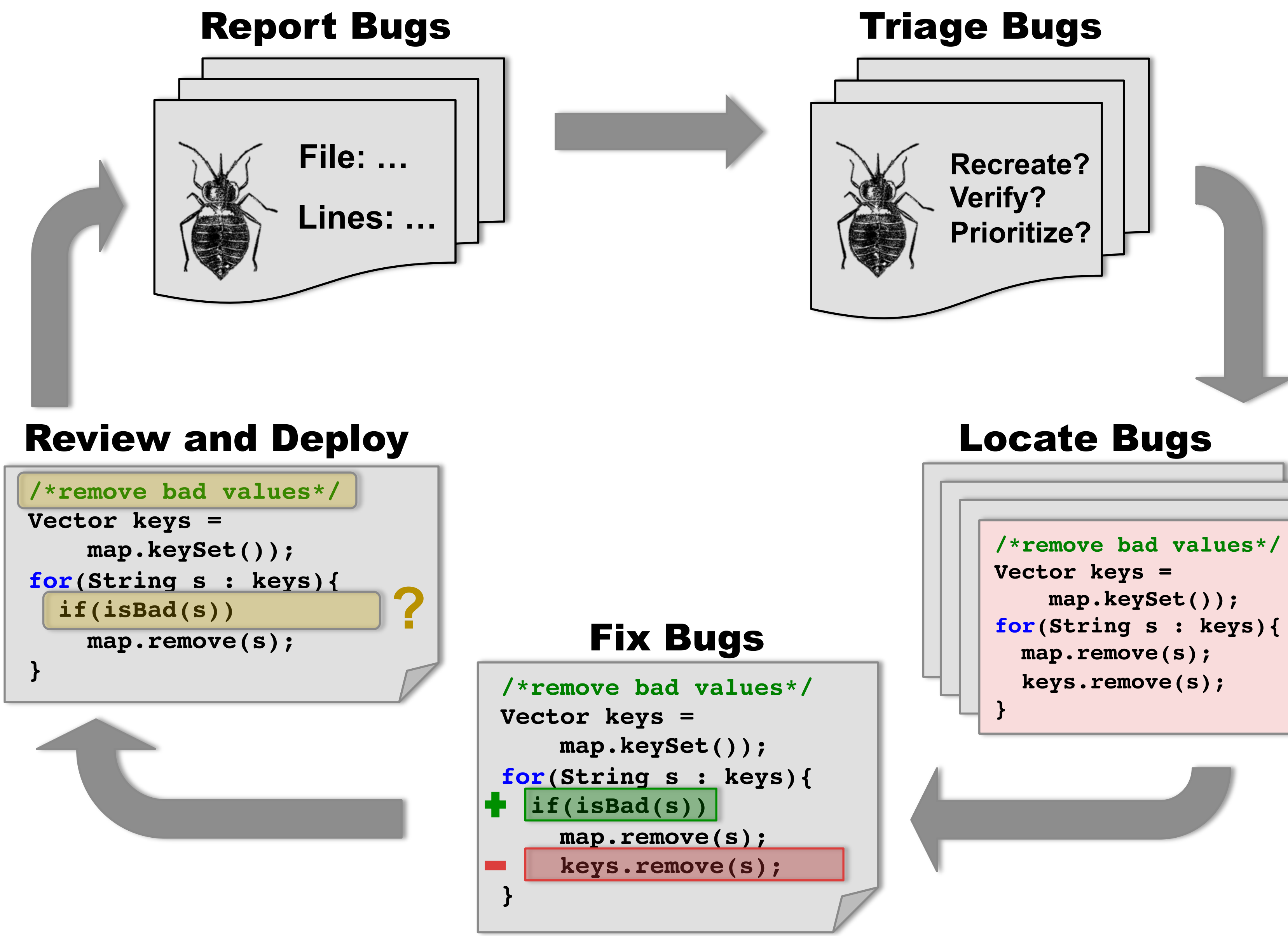
MAINTENANCE COSTS

Software maintenance can account for up to 90% of the software lifecycle costs.

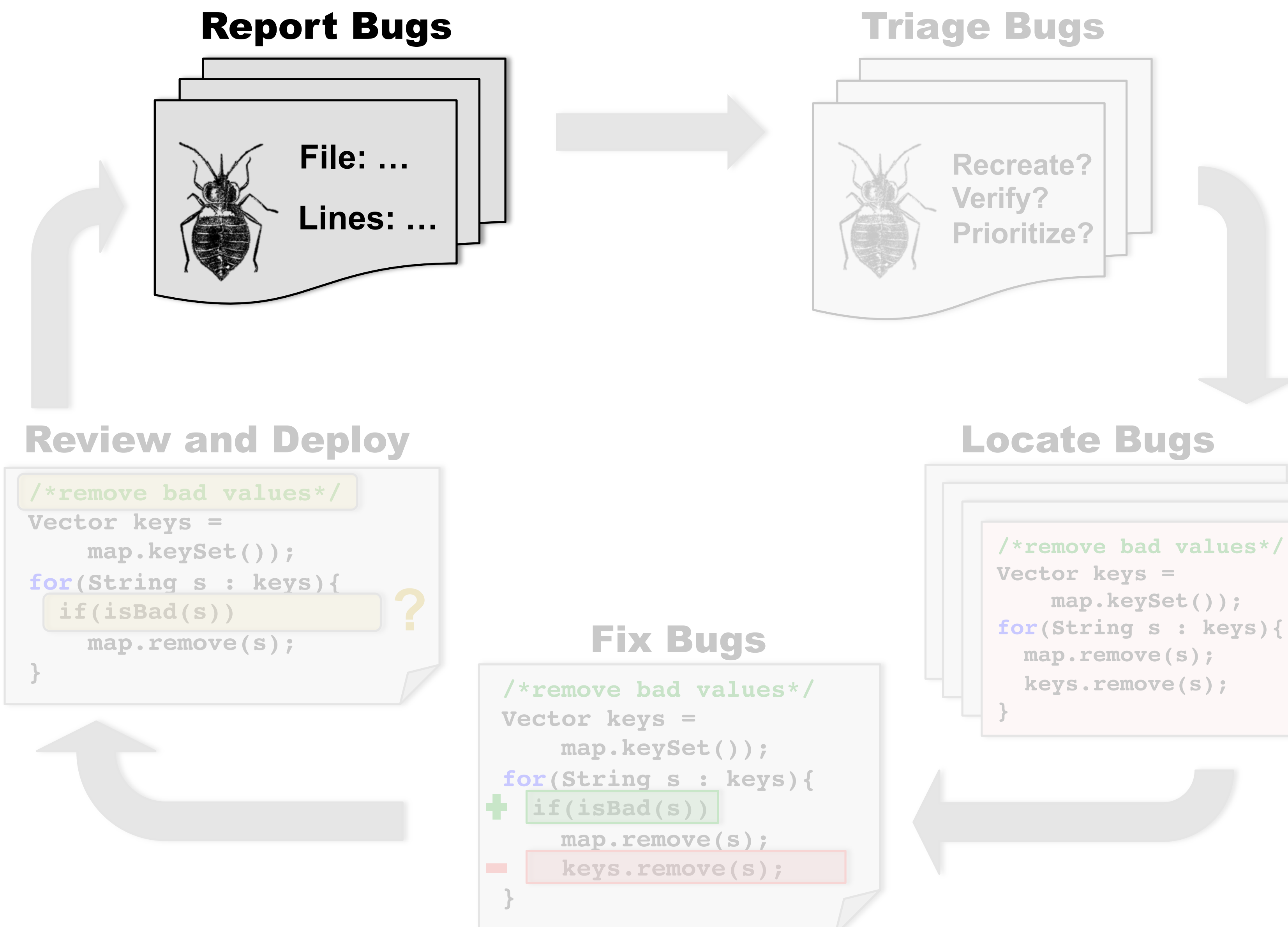


R.C. Seacord, D. Plakosh, and G. A. Lewis. Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 2003.

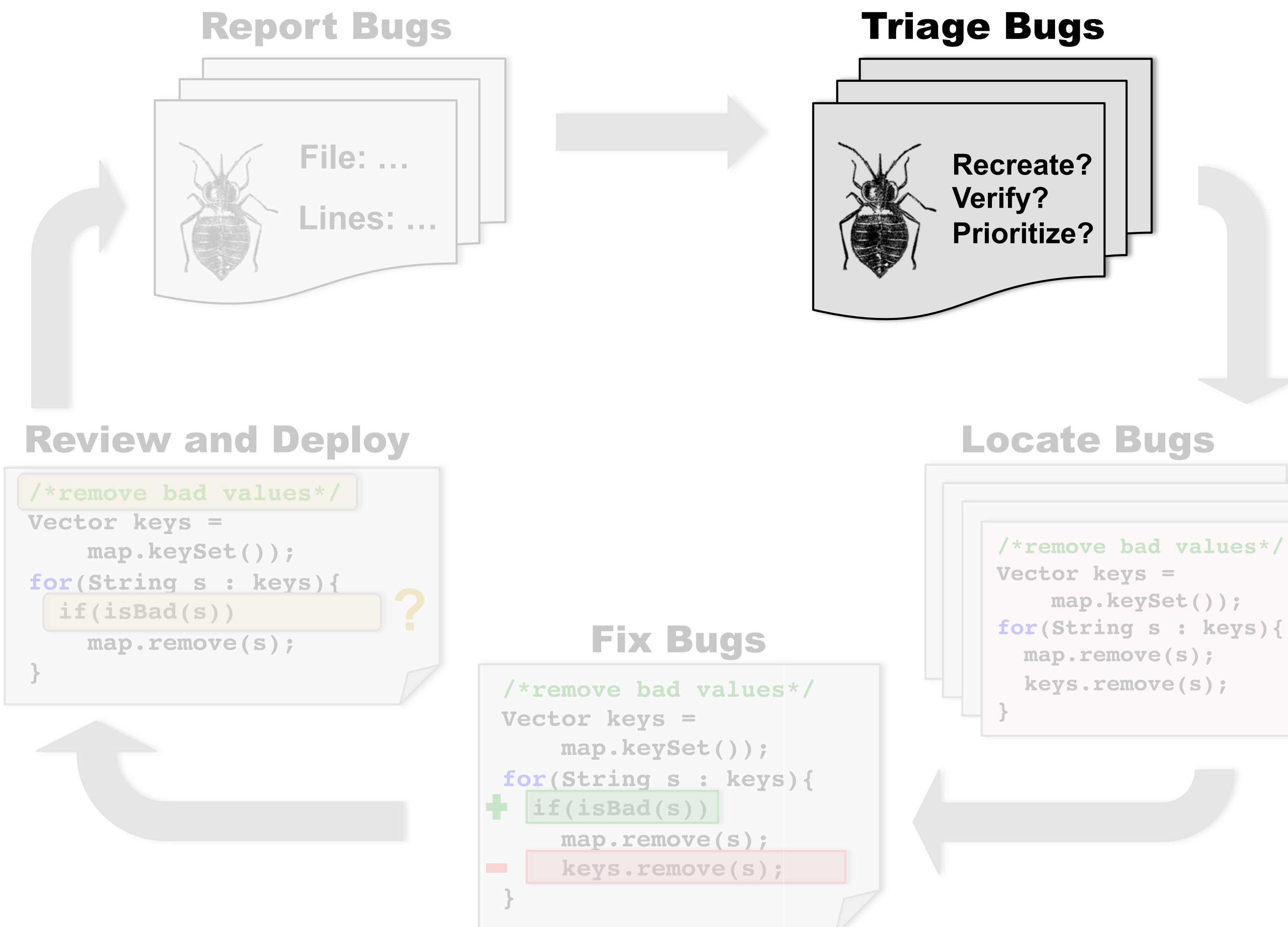
MAINTENANCE COSTS



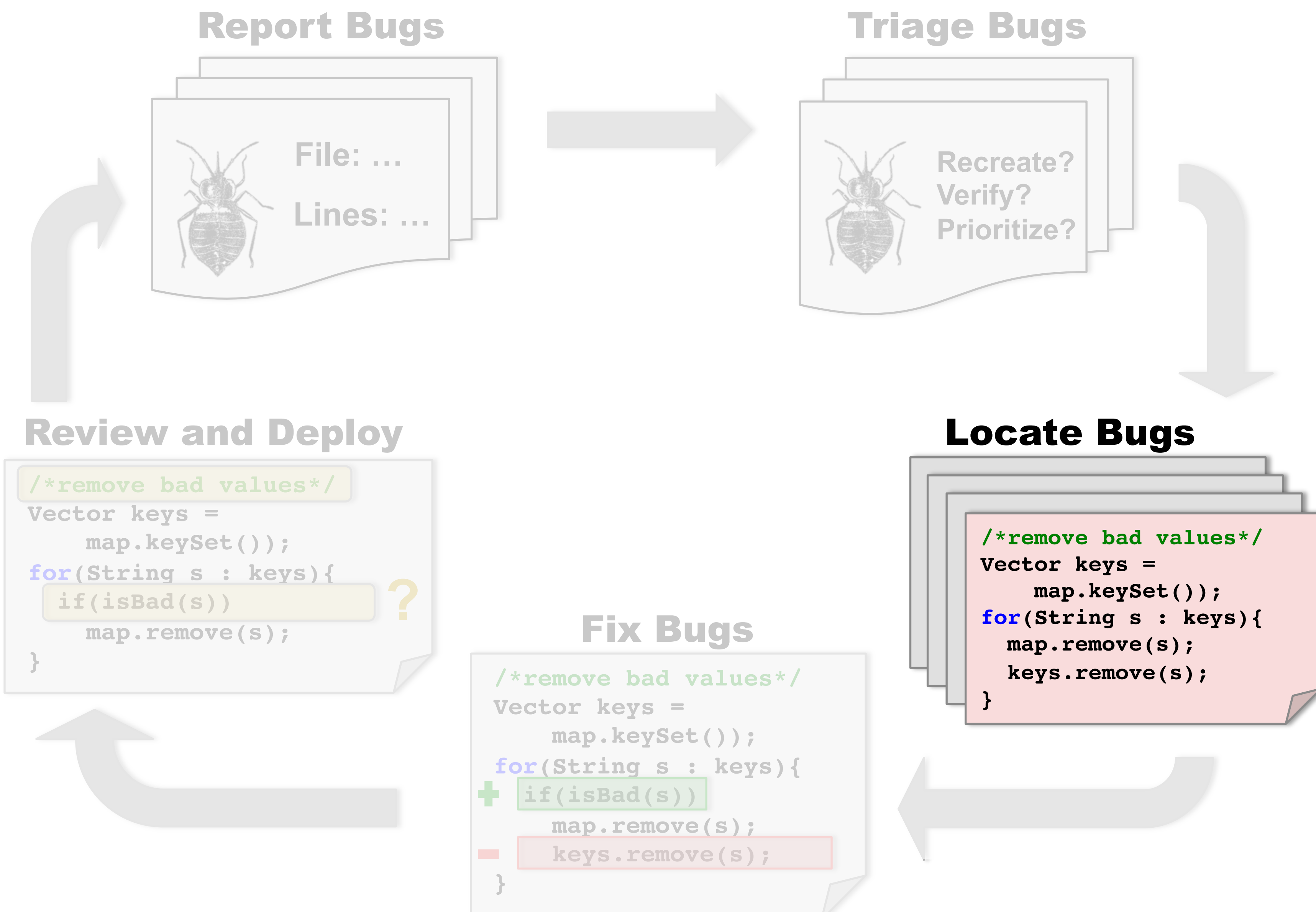
MAINTENANCE COSTS



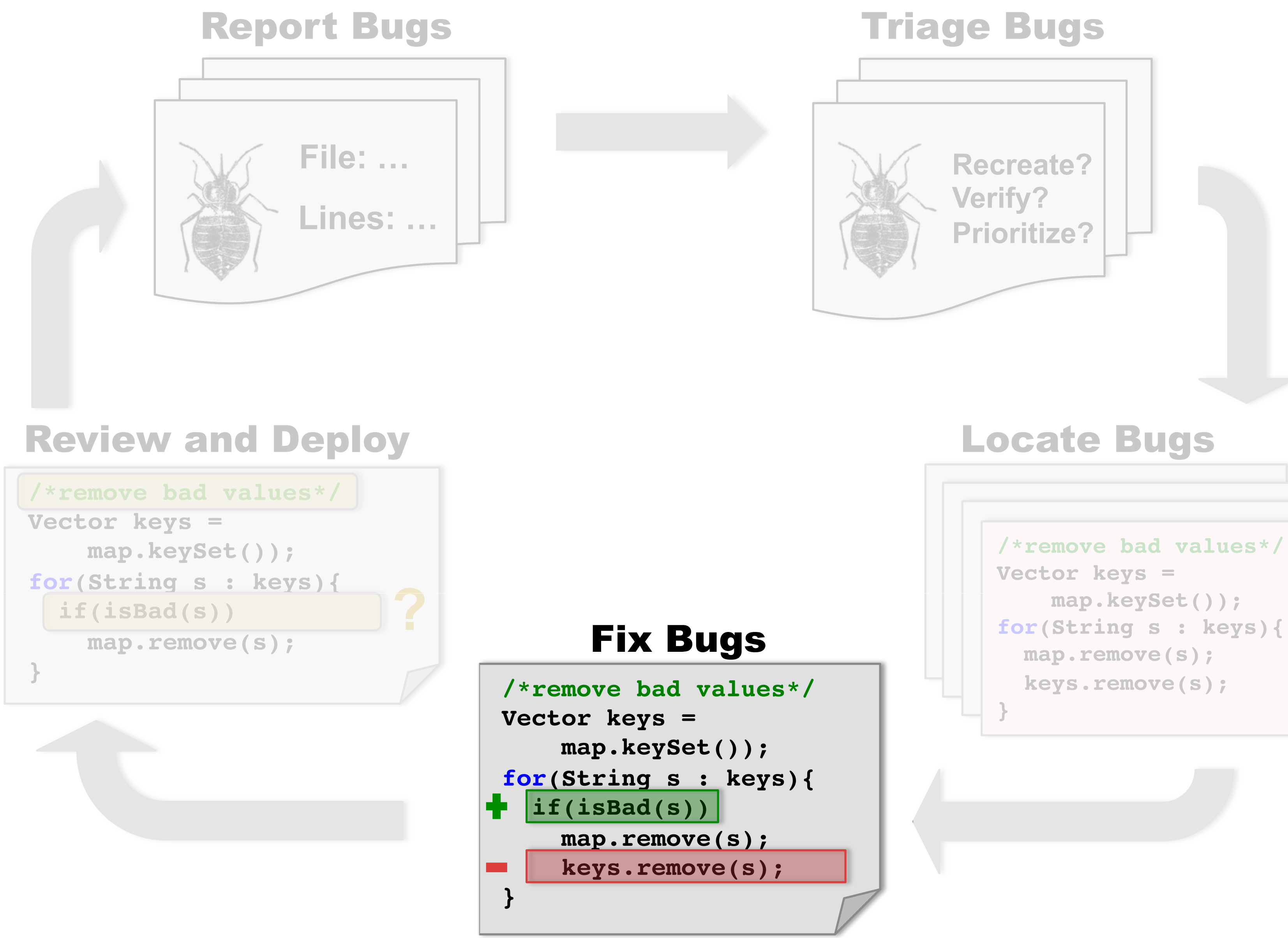
MAINTENANCE COSTS



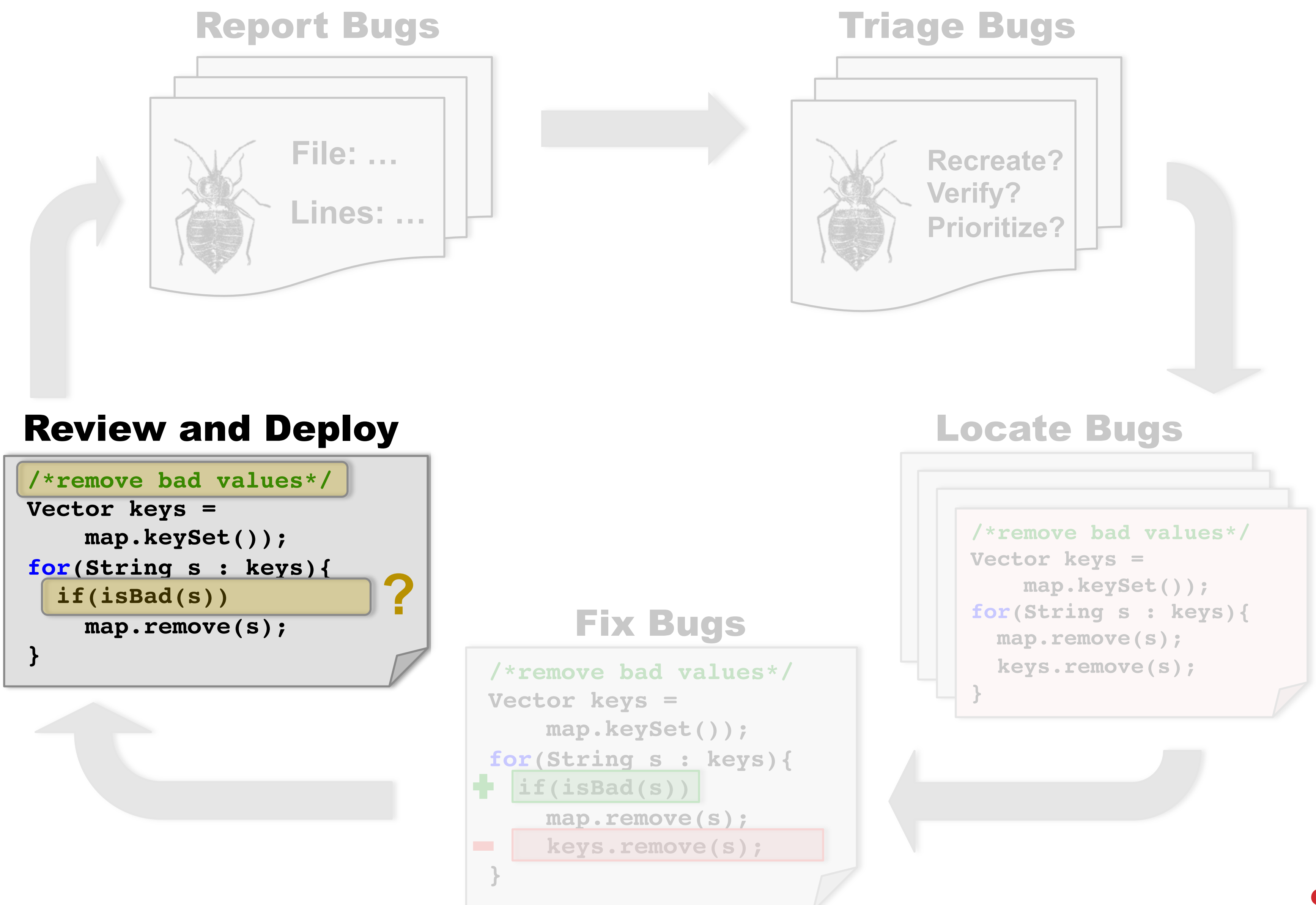
MAINTENANCE COSTS



MAINTENANCE COSTS



MAINTENANCE COSTS



SUMMARY

Add lightweight analyses to specific tasks to reduce the overall cost of software maintenance

1. Reducing triage/fix costs by clustering defect reports
2. Speeding up automatic patch generation technique
3. Showing that machine-generated patches are maintainable

MAINTENANCE PROCESSES IN PRACTICE

- **Automated techniques have helped.**
- **However, the process remains costly.**

Research question: Can we reduce the effort necessary for specific parts of the maintenance process, thereby reducing the overall cost?

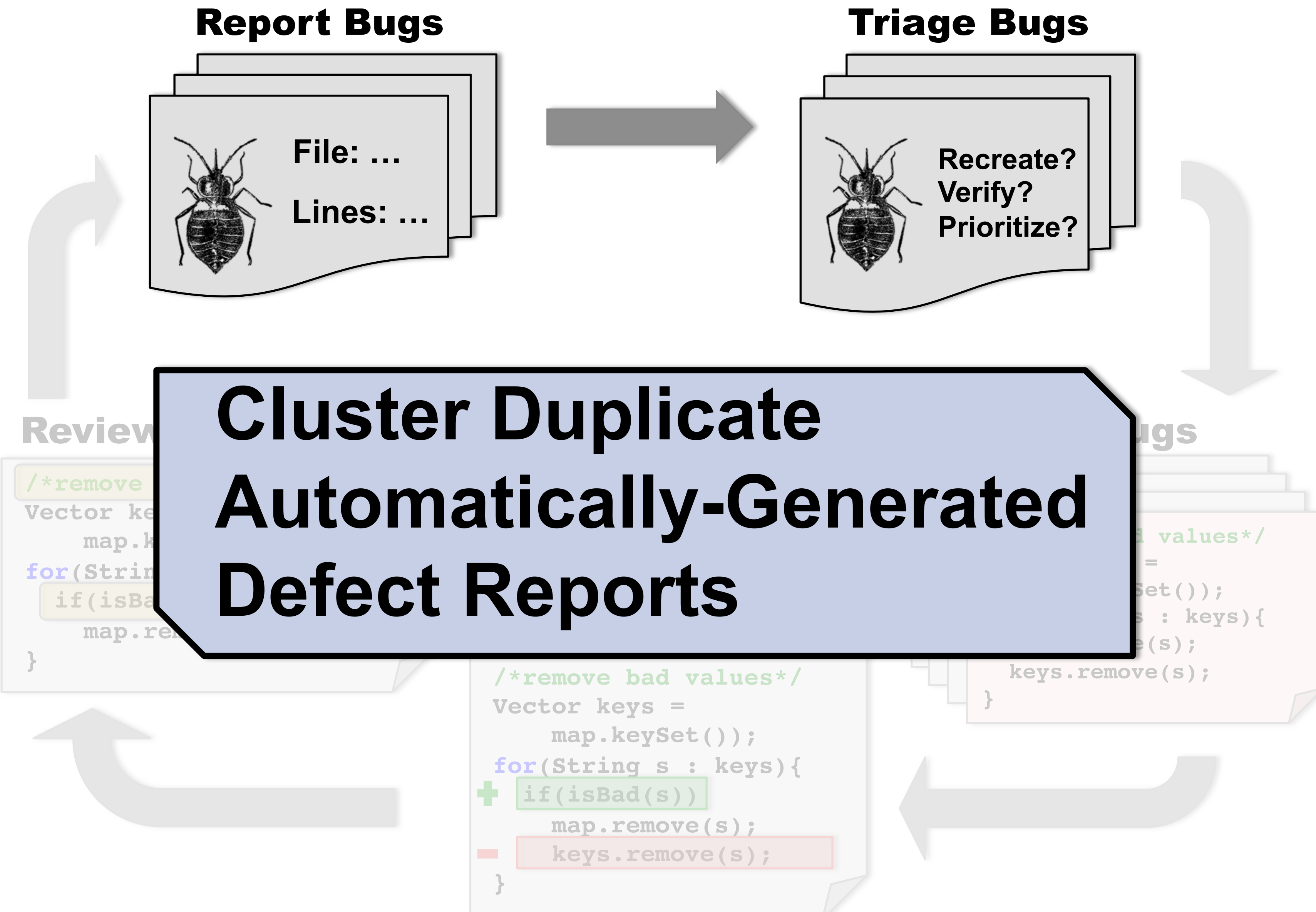
THESIS

Thesis: it is possible to construct usable and general light-weight analyses using both latent and explicit information present in software artifacts to aid in the finding and fixing of bugs, thus reducing costs associated with software maintenance in concrete ways.

RESEARCH CONSIDERATIONS

- **Generality**
  - Focus on a wide range of bugs to increase applicability
  - Could increase aggregate cost savings
- **Usability**
  - Minimize additional human effort
  - Ease incremental adoption
- **Comprehensive evaluation**
  - Traditional empirical success metrics
  - Human-centric notions of usability and quality

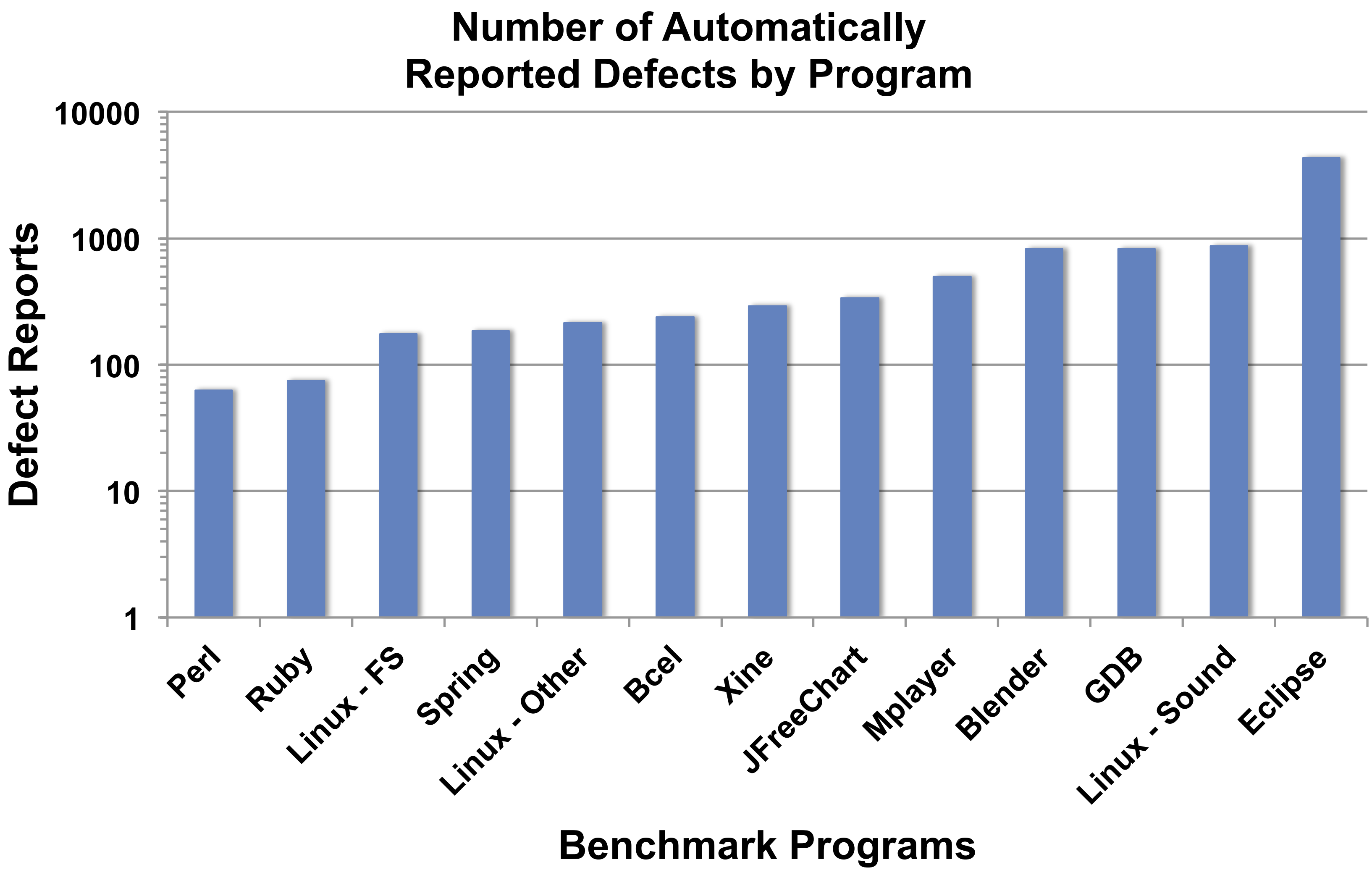
MAINTENANCE COSTS



AUTOMATIC BUG REPORTING IN PRACTICE

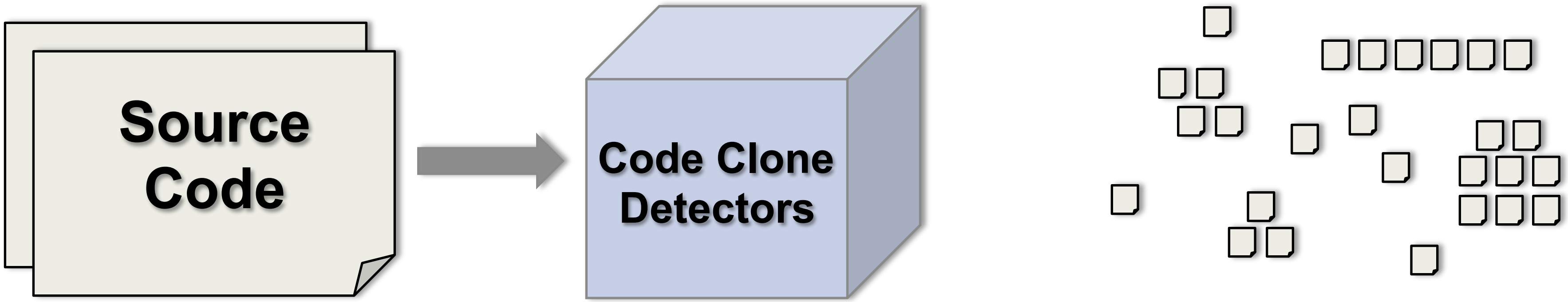
- **Manual bug reporting is costly**
  - Human effort
  - Direct and *indirect* costs
- **Automatic bug finders**
  - Help to find some bugs early
  - **Still require triage and fixing**
- **Goal: Cluster related defect reports to reduce subsequent human effort**

AUTOMATIC BUG REPORTING IN PRACTICE



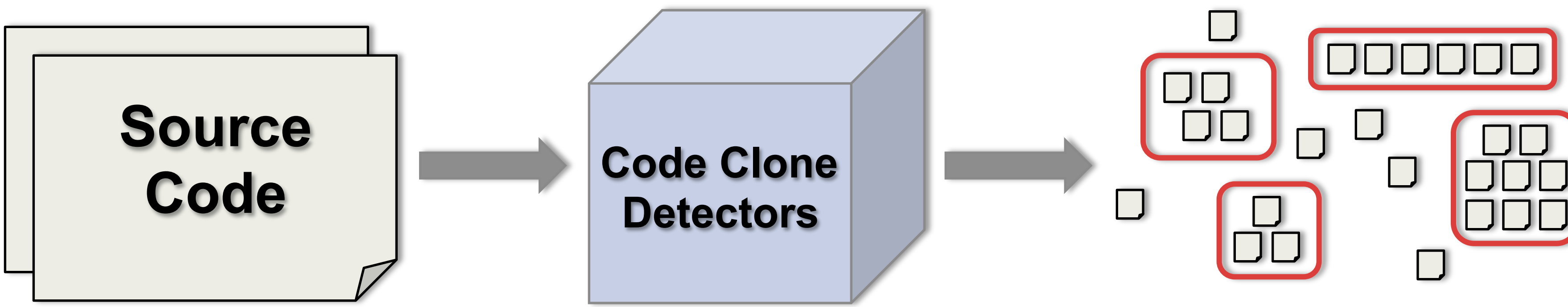
DUPLICATES IN GENERAL

Intuition: Duplicates are detrimental in related areas.



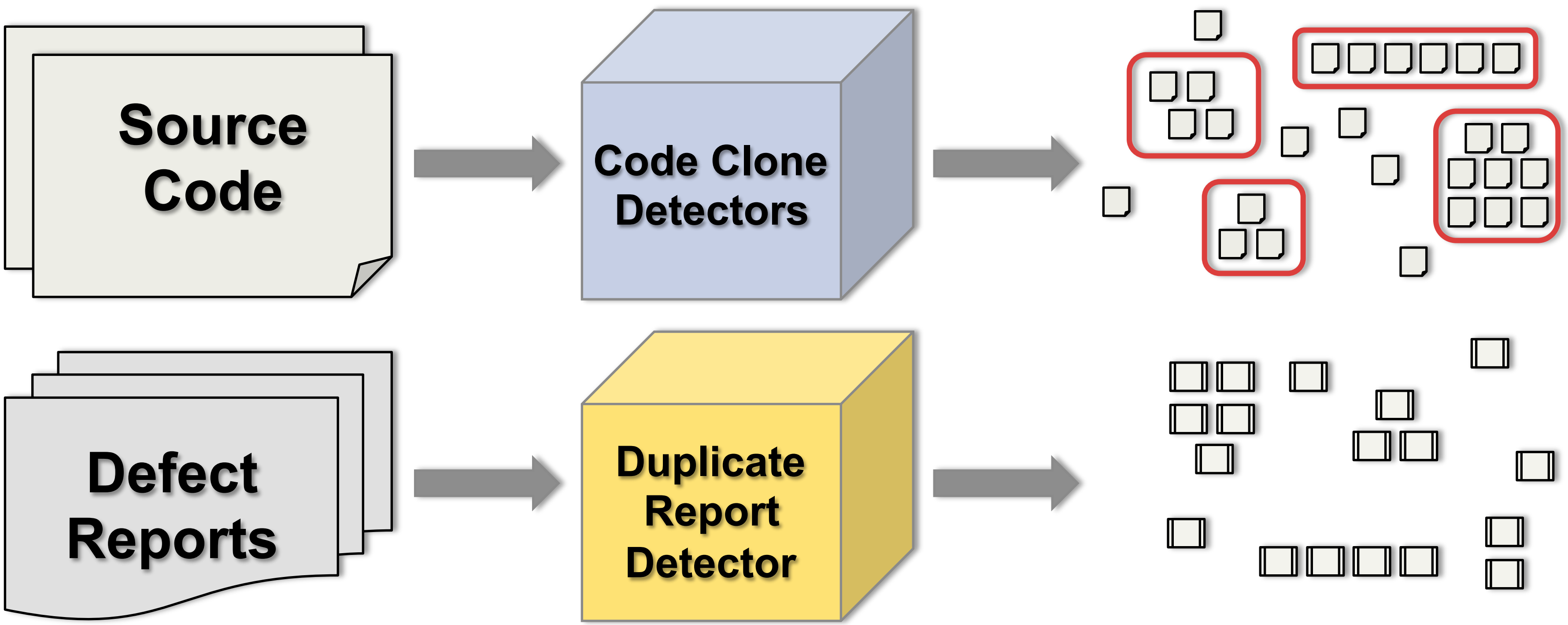
DUPLICATES IN GENERAL

Intuition: Duplicates are detrimental in related areas.



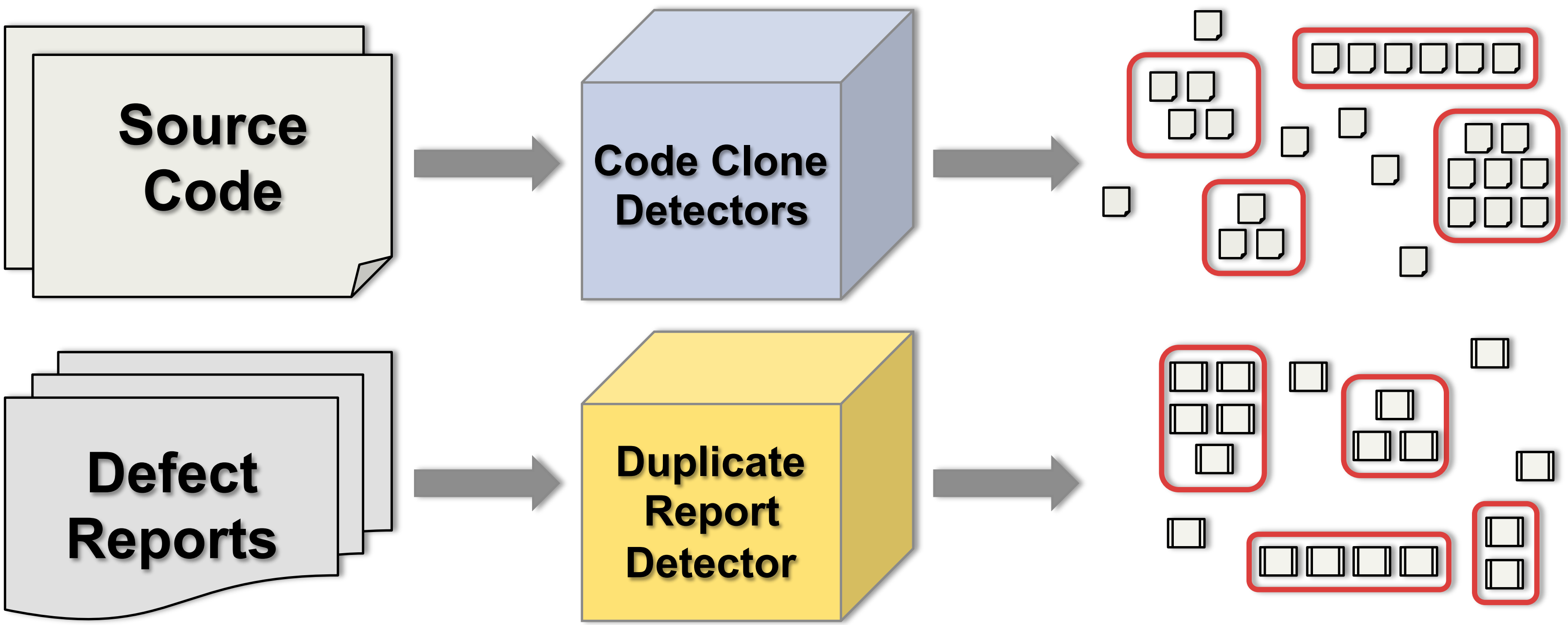
DUPLICATES IN GENERAL

Intuition: Duplicates are detrimental in related areas.



DUPLICATES IN GENERAL

Intuition: Duplicates are detrimental in related areas.



CLUSTERING: OVERVIEW

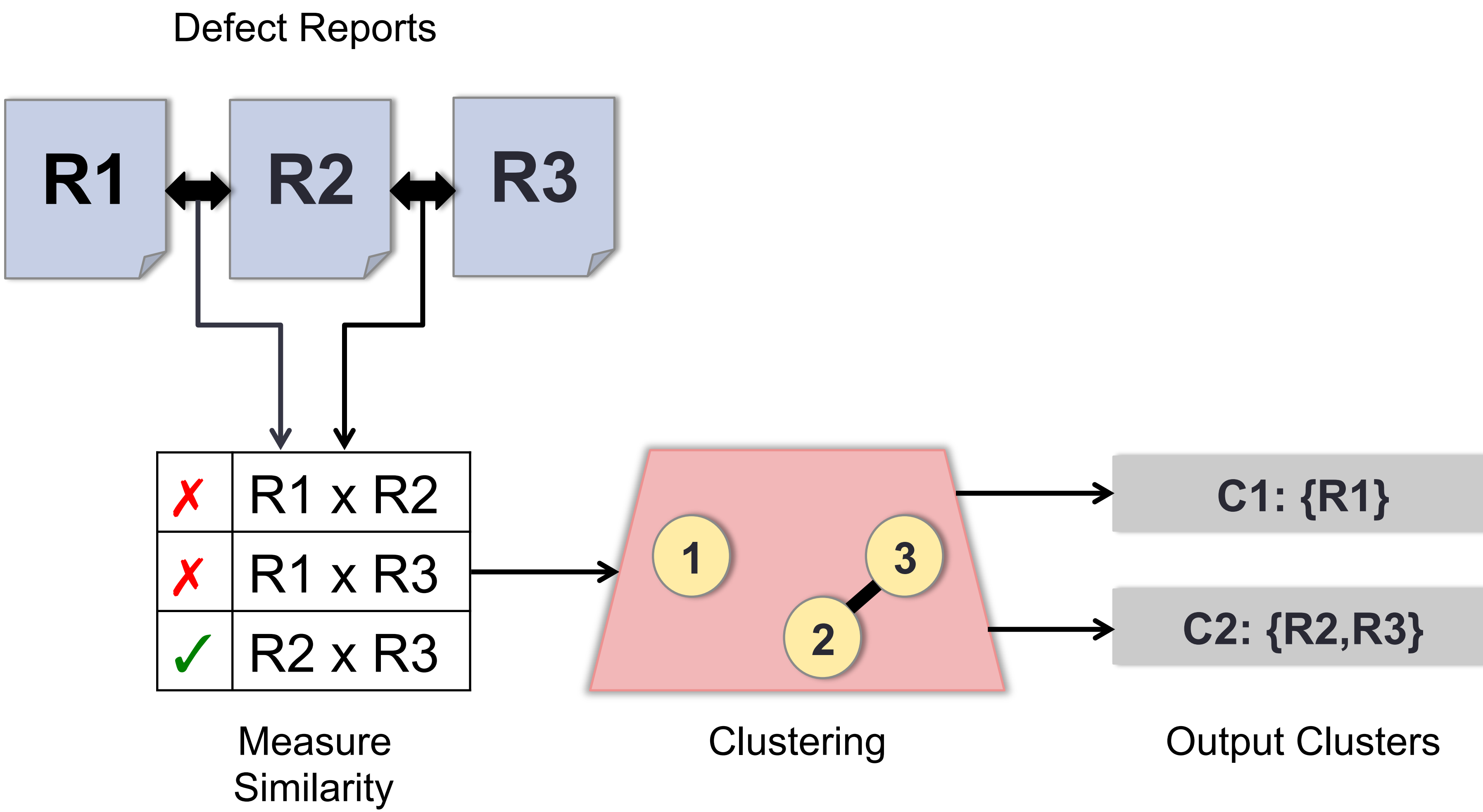
Goal: Cluster to reduce effort

Approach: Accurately cluster defect reports using structured comparison to save effort by handling similar defect reports in parallel.

Success depends on:

- Internal accuracy of the produced clusters
- Amount of effort saved from clustering defect reports

ALGORITHM OVERVIEW



STRUCTURED COMPARISON

Defect Report 1

File:

NSReader.java

Suspected Line:

```
plot = lst.get(i);
```

Defect Report 2

File:

NSReader.java

Suspected Line:

```
p = lst.get(i);
```

Defect Report 3

File:

NSReader.java

Suspected Line:

```
plot = lst.get(n);
```

Defect Report 4

File:

UI_Impl.java

Suspected Line:

```
plot = lst.get(i);
```

STRUCTURED COMPARISON

Defect Report 1

File:

NSReader.java

Suspected Line:

```
plot = lst.get(i);
```

Defect Report 2

File:

NSReader.java

Suspected Line:

```
p = lst.get(i);
```

Defect Report 3

File:

NSReader.java

Suspected Line:

```
plot = lst.get(n);
```

Defect Report 4

File:

UI_Impl.java

Suspected Line:

```
plot = lst.get(i);
```

STRUCTURED COMPARISON

Defect Report 1

File:

NSReader.java

Suspected Line:

```
plot = lst.get(i);
```

Defect Report 2

File:

NSReader.java

Suspected Line:

```
p = lst.get(i);
```

Defect Report 3

File:

NSReader.java

Suspected Line:

```
plot = lst.get(n);
```

Defect Report 4

File:

UI_Impl.java

Suspected Line:

```
plot = lst.get(i);
```

SIMILARITY METRICS

$$\frac{1}{(1 + \text{Levenshtein Edit Distance})}$$

```
plot = lst.get(i);

p = lst.get(i);

plot = lst.get(n);
```

SIMILARITY METRICS

$$\frac{1}{(1 + \text{Levenshtein Edit Distance})}$$

```
plot = lst.get(i);

p = lst.get(i);
```

$$\frac{1}{1+1} = \frac{1}{2}$$

```
plot = lst.get(n);
```

SIMILARITY METRICS

$$\frac{1}{(1 + \text{Levenshtein Edit Distance})}$$

```
plot = lst.get(i);
```

```
p = lst.get(i);

plot = lst.get(n);
```

$$\frac{1}{1+2} = \frac{1}{3}$$

STRUCTURED COMPARISON

Defect Report 1

File:

NSReader.java

Suspected Line:

```
plot = lst.get(i);
```

Defect Report 2

File:

NSReader.java

Suspected Line:

```
p = lst.get(i);
```

Defect Report 3

File:

NSReader.java

Suspected Line:

```
plot = lst.get(n);
```

Defect Report 4

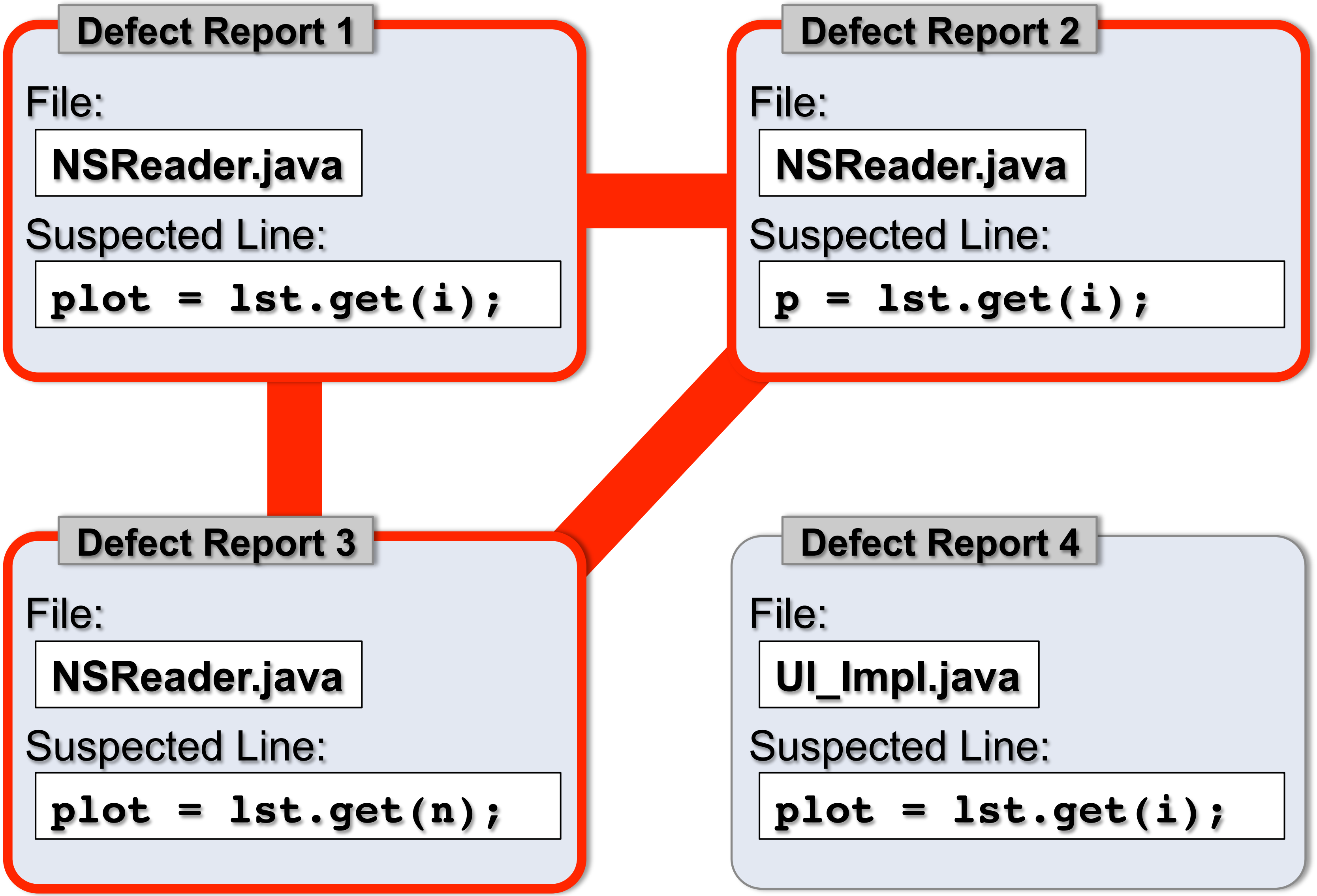
File:

UI_Impl.java

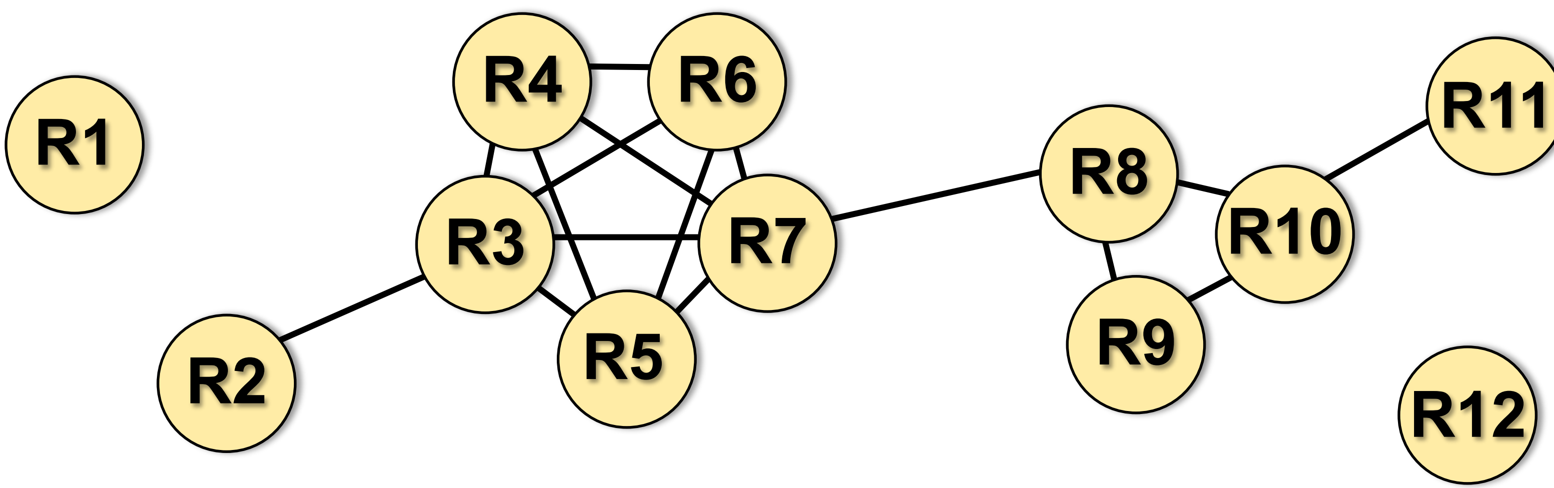
Suspected Line:

```
plot = lst.get(i);
```

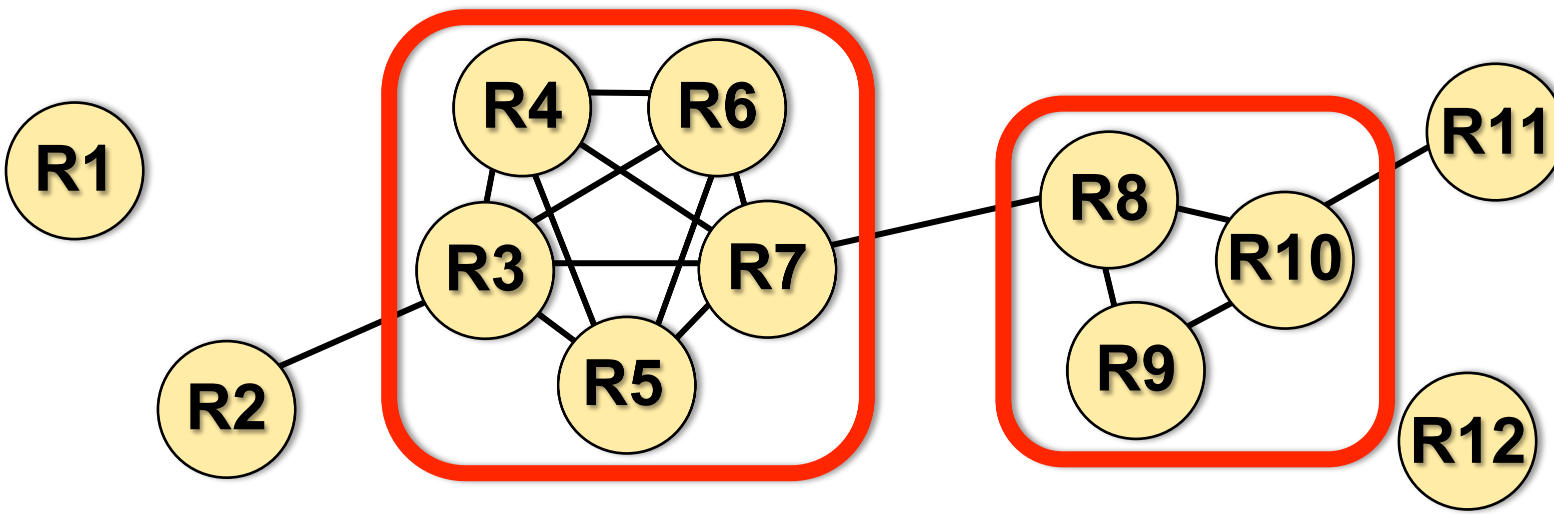
STRUCTURED COMPARISON



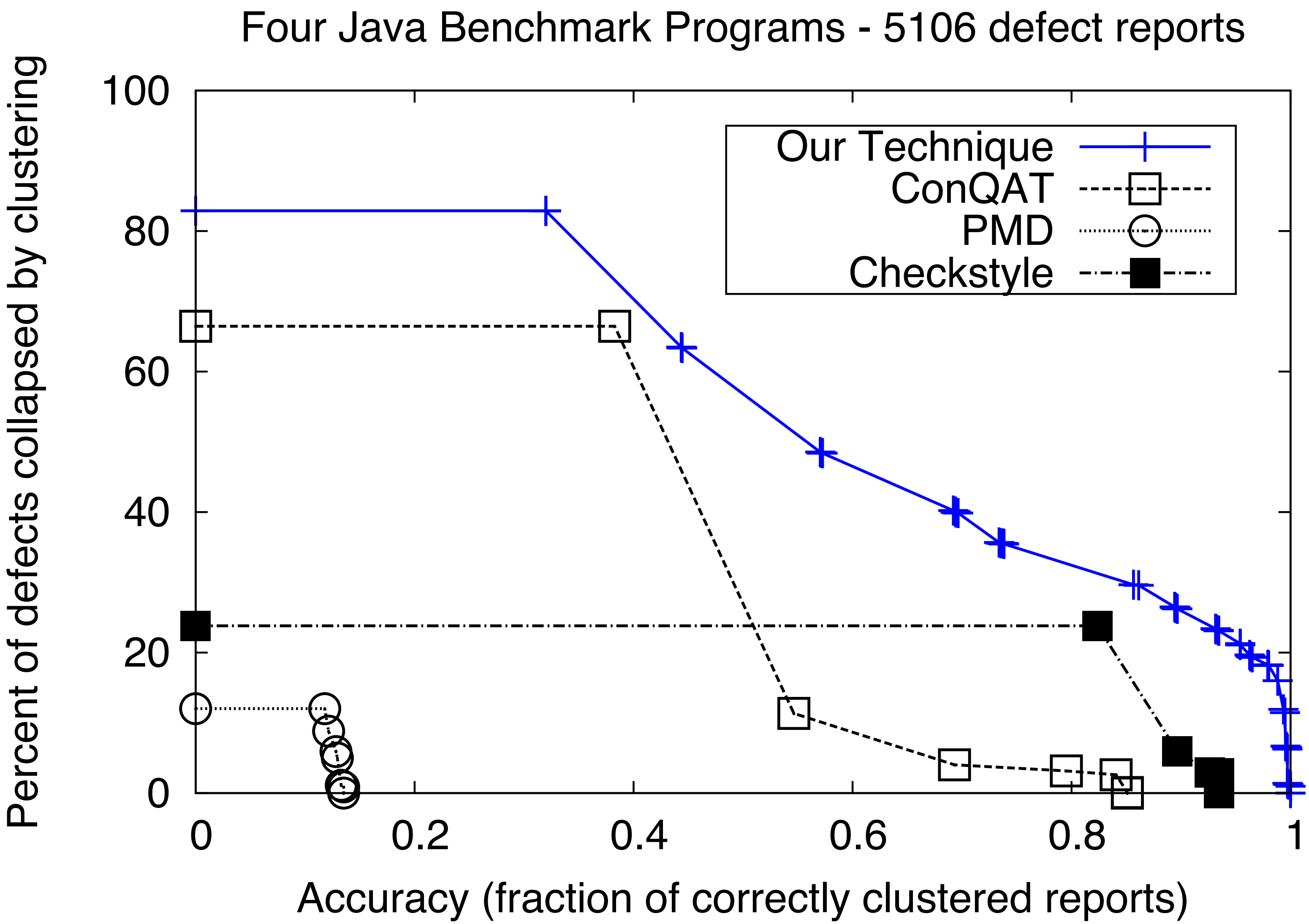
CLUSTERING TECHNIQUE



CLUSTERING TECHNIQUE

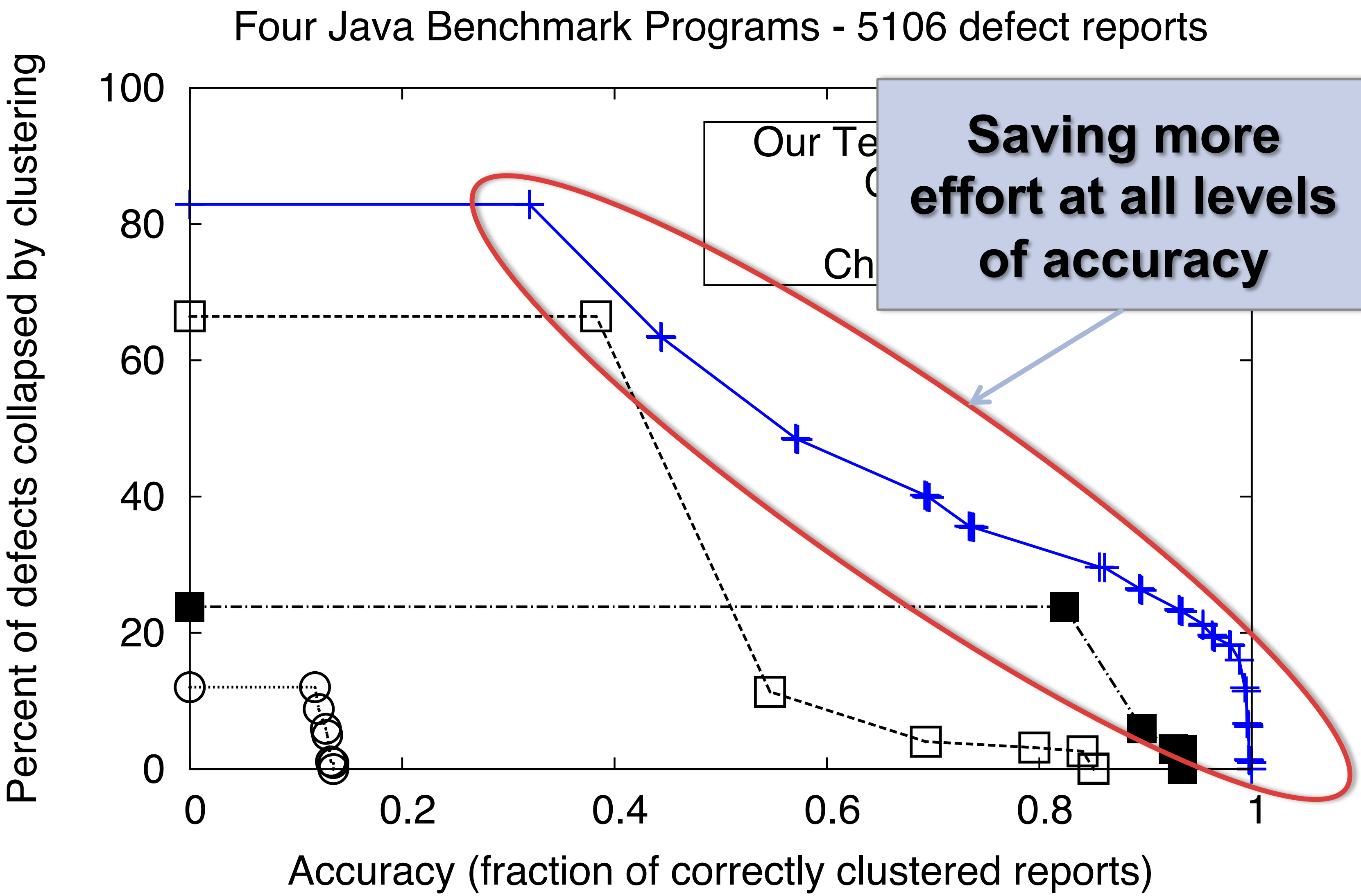


RESULTS

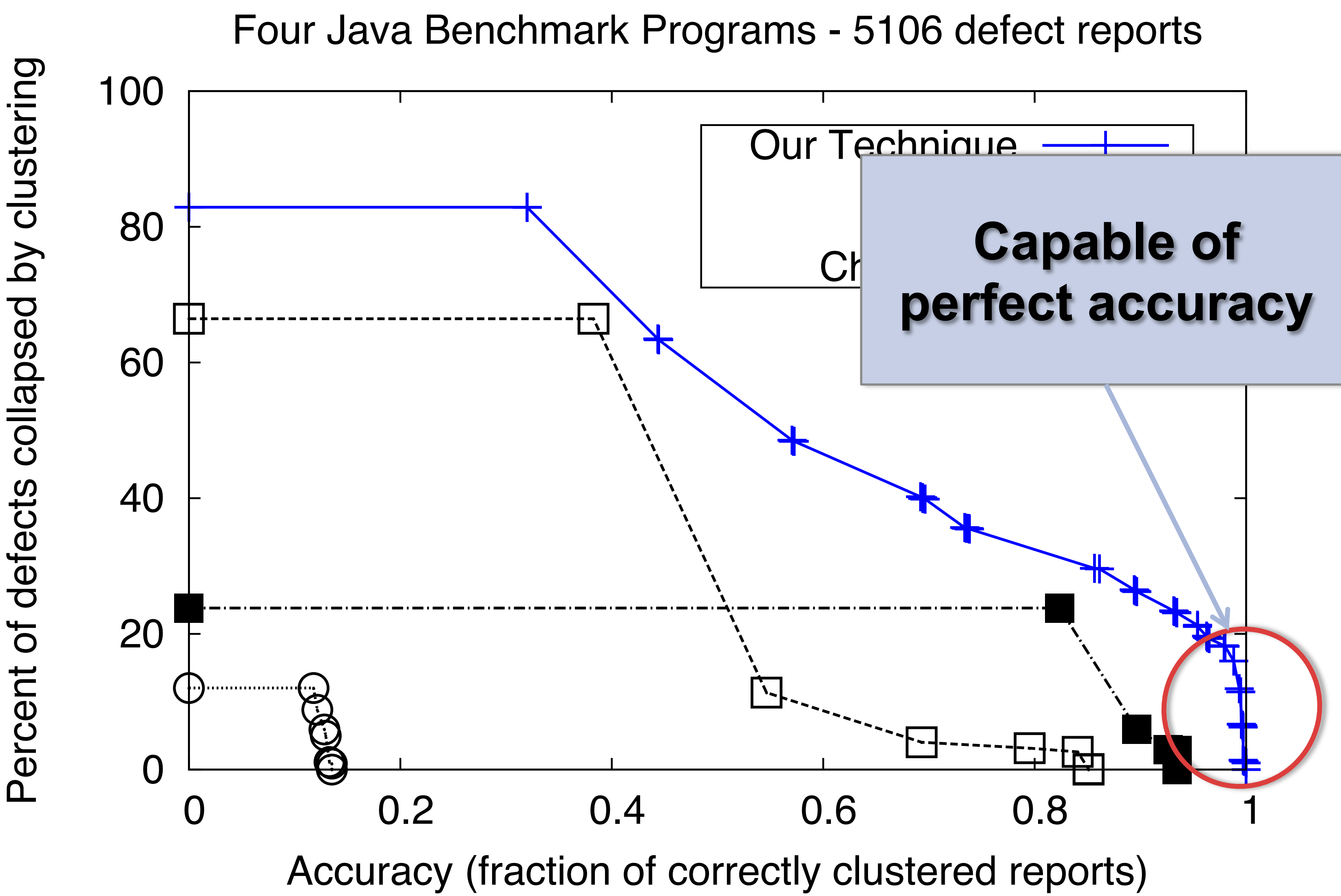


Four Java Benchmark Programs - 5106 defect reports
Percent of defects collapsed by clustering
100
80
60
40
20
0
0
0.2
0.4
0.6
0.8
1
Accuracy (fraction of correctly clustered reports)
Our Technique
ConQAT
PMD
Checkstyle

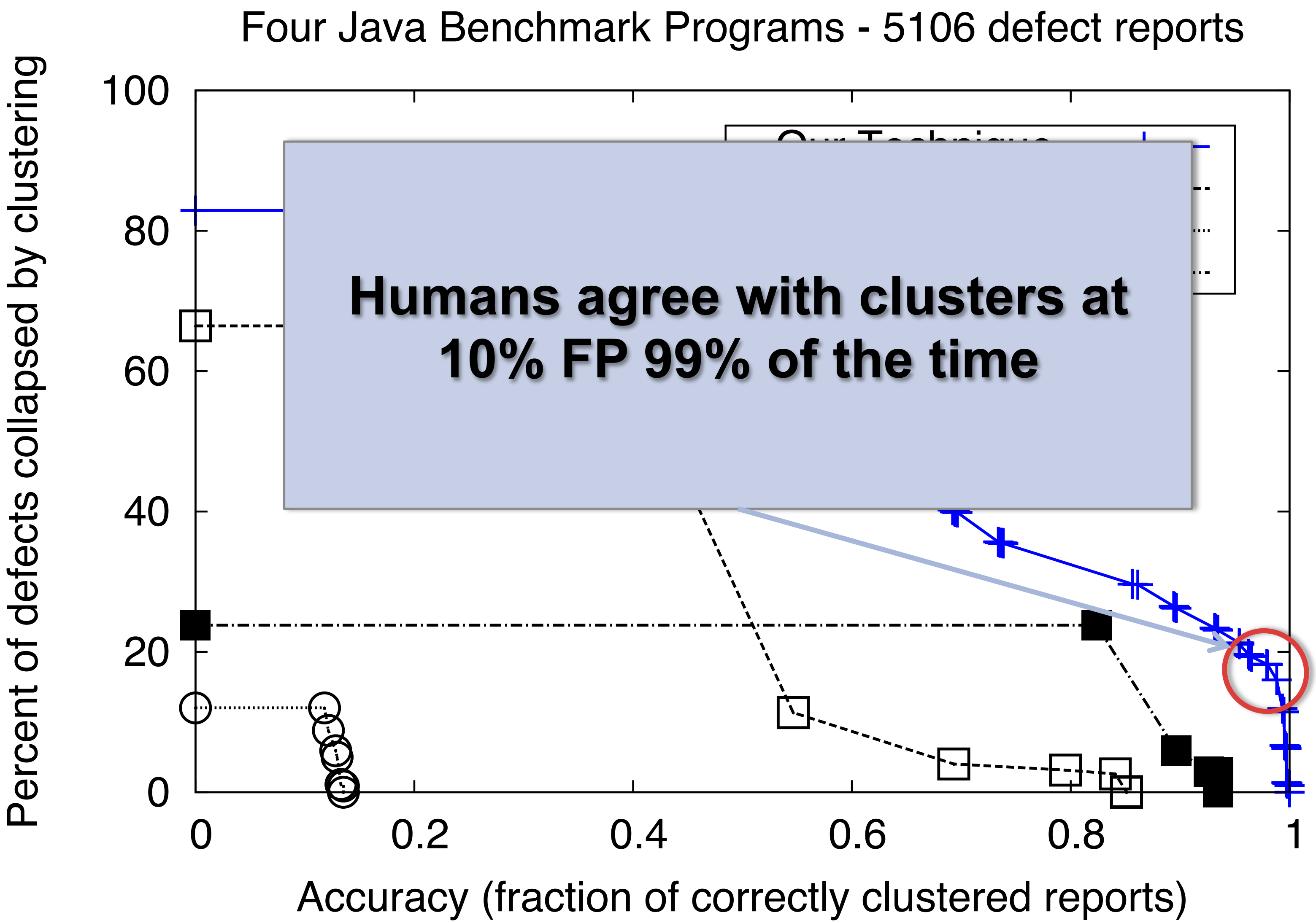
RESULTS


Four Java Benchmark Programs - 5106 defect reports
Percent of defects collapsed by clustering
Accuracy (fraction of correctly clustered reports)
0
20
40
60
80
100
0
0.2
0.4
0.6
0.8
1
Our Te
C
Ch
Saving more effort at all levels of accuracy

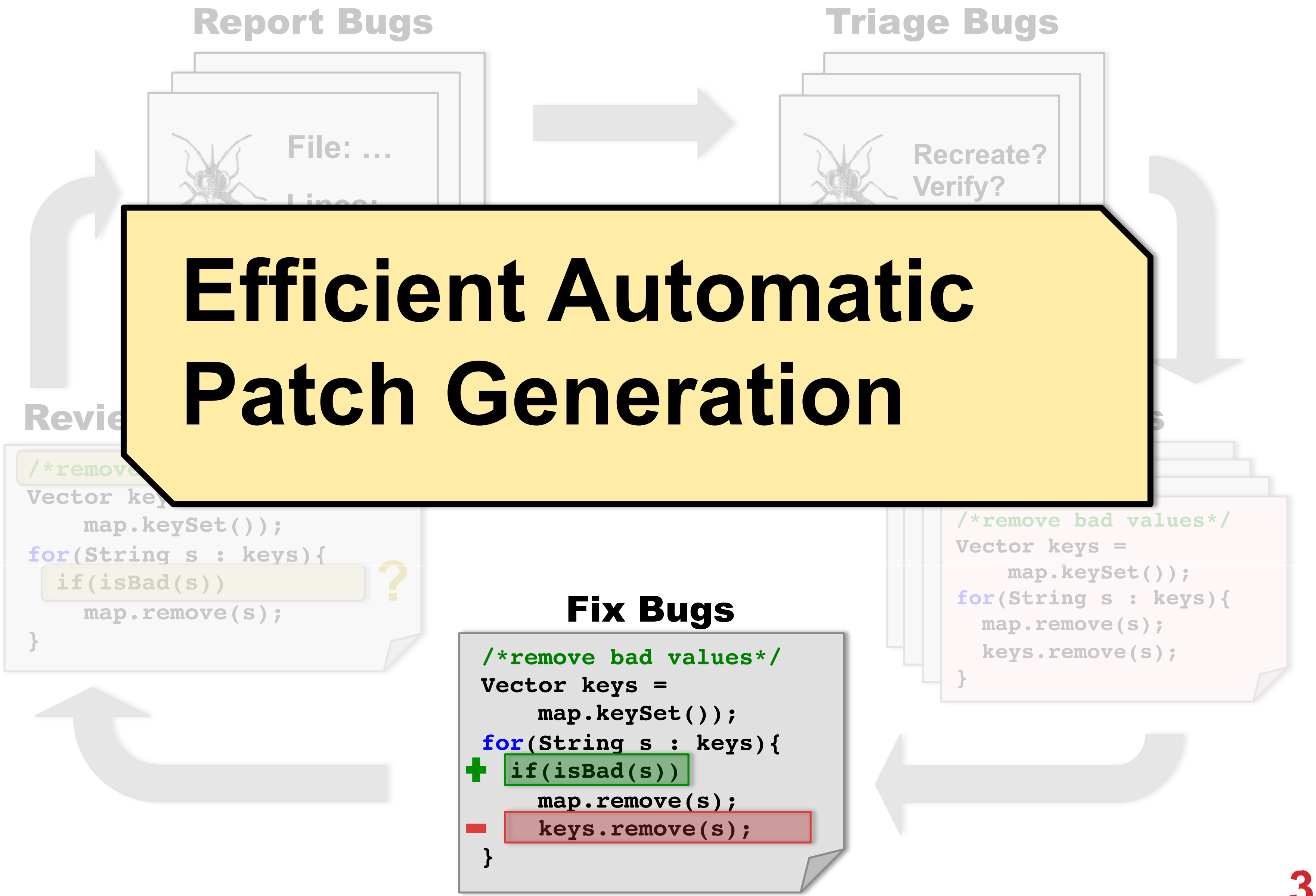
RESULTS


Four Java Benchmark Programs - 5106 defect reports
Percent of defects collapsed by clustering
100
80
60
40
20
0
0
0.2
0.4
0.6
0.8
1
Accuracy (fraction of correctly clustered reports)
Our Technique
Capable of perfect accuracy

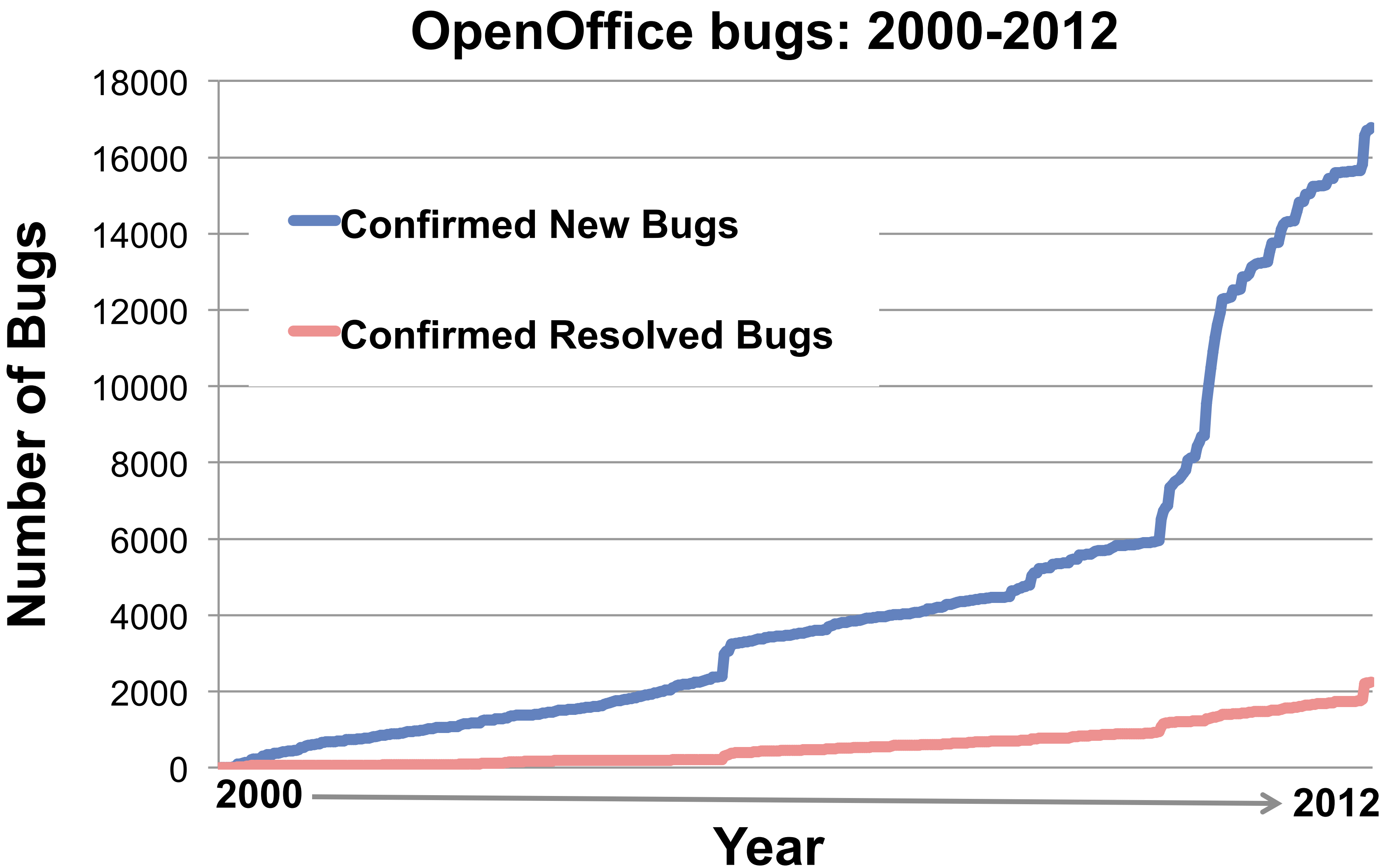
RESULTS


Four Java Benchmark Programs - 5106 defect reports
Humans agree with clusters at 10% FP 99% of the time
Percent of defects collapsed by clustering
Accuracy (fraction of correctly clustered reports)
0
20
40
60
80
100
0
0.2
0.4
0.6
0.8
1

MAINTENANCE COSTS



BUGS VS. FIXES



AUTOMATIC PROGRAM REPAIR

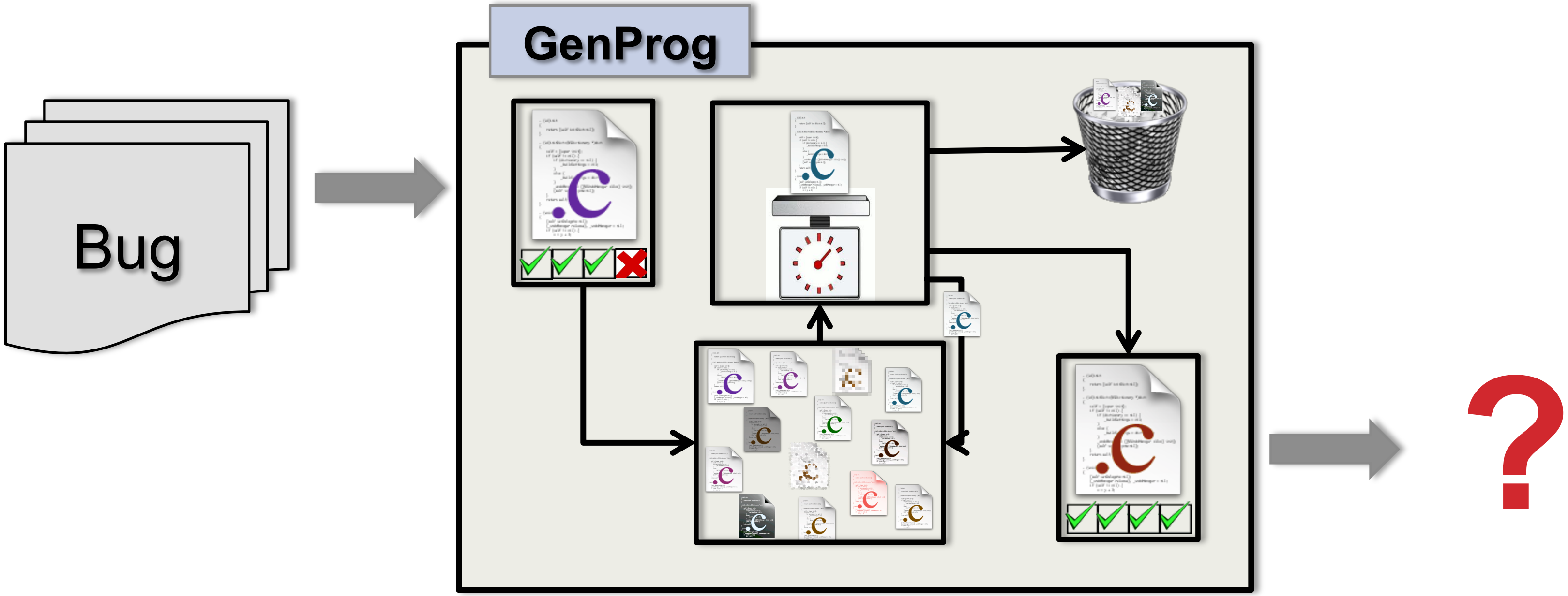
Current manual fix strategies fail to keep up with bug reporting rates

Automatic program repair techniques show promise

- GenProg – genetic algorithm-based patch generation

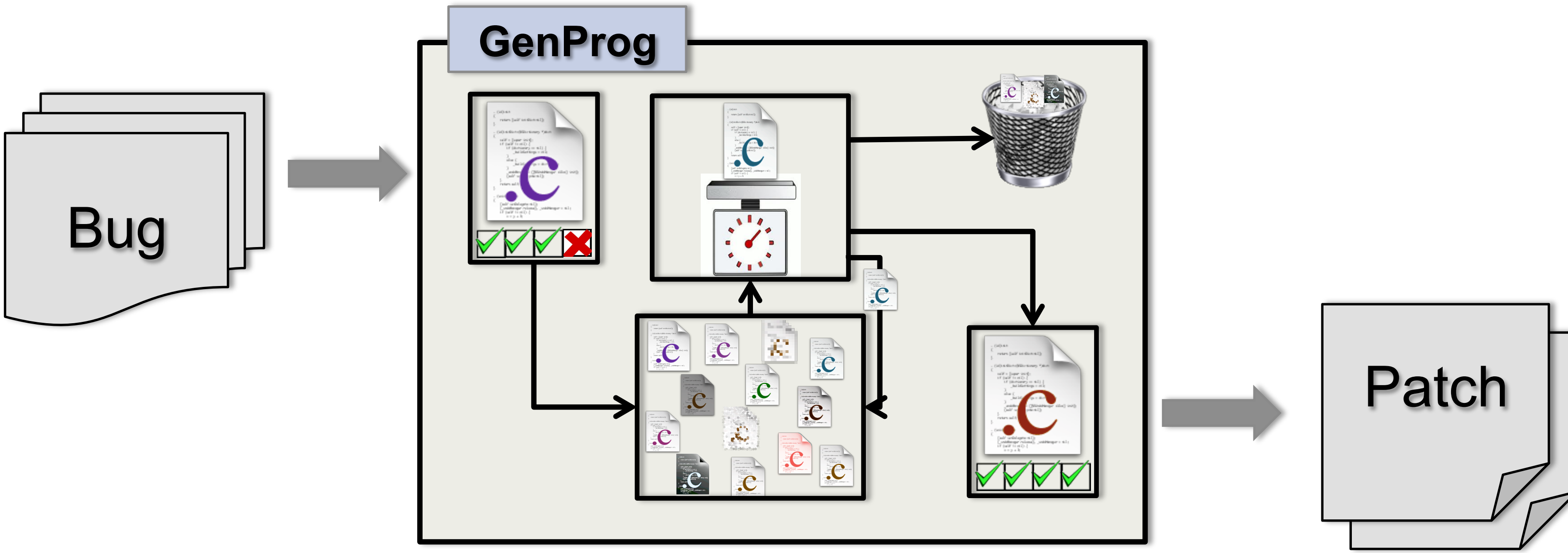
GENPROG ARCHITECTURE

Automatic program repair



GENPROG ARCHITECTURE

Automatic program repair can generate patches.



However, sometimes long fixes and high variance

PROGRAM REPAIR: OVERVIEW

- Learning from past results
  - Syntactically different changes often yield identical behavior
  - Certain tests fail more often when making changes to specific parts of a program
- Intuitions
  - Evaluating semantically identical changes is redundant
  - Adaptive online learning can drive a “fail early” test selection strategy

PROGRAM REPAIR: OVERVIEW

Approach: quotient the search space semantically and use historical data to efficiently test potential patches up to a given size.

"AE" – Adaptive + Equivalence

Success depends on:

- Concrete improvement of internal cost metrics
  - Time spent testing
  - Number of patches considered
- Dollar cost

APPROACH: OVERVIEW

Quotient the search space of program changes based on semantic meaning

- Identify classes of equivalent patches and avoid checking them redundantly

Dual of mutation testing

MUTATION TESTING

- Goal: Measure test suite adequacy
- Approach: Mutate a program to simulate bugs, then measure how many changes a test suite exposes
- Problem: Equivalent mutants – false adequacy penalty (correctness)

Using similar analyses, we find equivalent patches as an optimization

APPROACH: EXAMPLE

Quotient the search space to avoid repeating work

```
C=100;
```

```
A=1;
B=2;
C=3;
D=4;
return A,B,C,D;
```

APPROACH: EXAMPLE

Quotient the search space to avoid repeating work

```
C=100;
```

```
A=1;
B=2;
C=3;
D=4;
return A,B,C,D;
```

APPROACH: EXAMPLE

Test Prioritization

- **Use historical feedback for testing**

Mutant X

Test	Failures
T1	0
T2	1
T3	15
T4	7
T5	0

Strategy
1. **T3**
2. T4
3. T2
4. T1
5. T5

ALGORITHM

My approach – exhaustively search all potential single-edit patches

```
For every patch:
    if patch equivalent to previous:
        continue;
    For every test:
        Run patch on test
        if test fails
            break;
    Return if patch passes all tests
```

ALGORITHM: IMPROVEMENTS

My approach – Exhaustively search all potential single-edit patches

```
For every patch:
    if patch equivalent to previous:
        continue;
    For every test:
        Run patch on test
        if test fails
            break;
    Return if patch passes all tests
```

Quotient Search Space

Test Prioritization

ALGORITHM: IMPROVEMENTS

My approach – Exhaustively search all potential single-edit patches

```
For every patch:
    if patch equivalent to previous:
        continue;
    For every test:
        Run patch on test
        if test fails
            break;
    Return if patch passes all tests
```

Quotient Search Space

Test Prioritization

↓ 88.3%

↓ 94.3%

IMPROVEMENTS: COST

Monetary Cost

- Tried to patch 105 bugs on Amazon's EC2
- 70.2% cost reduction compared to previous GenProg results
  - Both techniques patched roughly 50% of the available bugs
  - Fixed costs (equivalence analysis)

MAINTENANCE COSTS

A Human Study of Patch Maintainability

Report Bugs

File: …

Lines:

Triage Bugs

Locate Bugs

```
/*remove bad values*/
Vector keys =
    map.keySet());
for(String s : keys){
  map.remove(s);
  keys.remove(s);
}
```

Fix Bugs

```
/*remove bad values*/
Vector keys =
    map.keySet());
for(String s : keys){
+ if(isBad(s))
    map.remove(s);
-   keys.remove(s);
}
```

Review and Deploy

```
/*remove bad values*/
Vector keys =
    map.keySet());
for(String s : keys){
  if(isBad(s))
    map.remove(s);
}
```

?

BUG FIXING AND MAINTAINABILITY

- **AE and GenProg can fix about 50% of bugs**
- **Saves human effort for current bugs**
- **We want to test the future maintainability of patches to evaluate various techniques' efficacies over time**
  - Can automatically-generated patches help reduce the maintenance debt?

CODE QUALITY

Functional Quality

- Does the implementation pass the supplied test suite?
- Does the code execute “correctly”?

Non-functional Quality

- Is the code understandable to humans?
- How difficult is it to alter the code in the future?

CODE QUALITY

Functional Quality ✓

- Does the implementation pass the supplied test suite?
- Does the code execute "correctly"?

Non-functional Quality ?

- Is the code understandable to humans?
- How difficult is it to alter the code in the future?

GOALS – QUESTIONS

1. How can we concretely measure these notions of human understandability and future maintainability?
2. In practice, are machine-generated patches as maintainable as human-generated patches?
3. Can we automatically augment machine-generated patches to improve maintainability?

SUCCESS CRITERIA

Approach:

- Find a method for concretely measuring human maintainability
- Evaluate various types of patches

Success depends on:

- Providing evidence that automatically-generated patches can be as maintainable as those created by humans

MEASURING MAINTAINABILITY

Indirect software quality metrics:

- Cyclomatic complexity
- Coupling and cohesion
- Software readability

MEASURING MAINTAINABILITY

Indirect software quality metrics:

- Cyclomatic complexity
- Coupling and cohesion
- Software readability

Direct measures of maintainability:

- Rather than using an approximation, we will directly measure humans' abilities to perform maintenance tasks

MEASURING MAINTAINABILITY

- **Goal: directly simulate the maintenance process**
- **Solution: ask human participants questions that require them to read and understand a piece of code and measure accuracy and effort**
  - Sillito *et al.* – "Questions programmers ask during software evolution tasks"

SUBJECT TASKS

- **Sillito *et al.* – "Questions programmers ask during software evolution tasks"**
  - Recorded and categorized the questions developers actually asked while performing real maintenance tasks
- **Example: "What is the value of the variable 'y' on line *X*?"**
  - Narrowly focused on the sight of the patch itself
  - Not: "Does this type have any siblings in the type hierarchy?"

TYPES OF PATCHES

- Original – the defective, un-patched original code used as a baseline
- Human-Accepted – human patches that have not been reverted to date
- Machine – automatically-generated patches (by GenProg or AE-type tool)
- Machine+Doc – the same machine-generated patches as above, but augmented with automatically synthesized documentation

TYPES OF PATCHES

- **Original** – the defective, un-patched original code used as a baseline
- **Human-Accepted** – human patches that have not been reverted to date
- **Machine** – automatically-generated patches (by GenProg or AE-type tool)
- **Machine+Doc** – the same machine-generated patches as above, but augmented with automatically synthesized documentation

AUTOMATIC DOCUMENTATION

- **Human patches may contain comments with hints about developer intention**
  - Automatic approaches cannot easily reason about ***why*** a change is made, but can describe ***what*** was changed

AUTOMATIC DOCUMENTATION

- **Human patches may contain comments with hints about developer intention**
  - Automatic approaches cannot easily reason about ***why*** a change is made, but can describe ***what*** was changed
- **Automatically Synthesized Documentation:**
  - We adapt DeltaDoc (Buse *et al*. ASE 2010)
  - Measures semantic program changes
  - Outputs natural language descriptions of changes

HUMAN STUDY TASKS

```
…
if (dc->prev) {
   if (con->conf.log_condition_handling) {
       log_error_write(srv, __FILE__, __LINE__,
      "sb", "go prev", dc->prev->key);
    }
    /* make sure prev is checked first */
    config_check_cond_cached(srv, con, dc->prev);
    /* one of prev set me to FALSE */
    if (COND_RESULT_FALSE == con->cond_cache[dc->context_ndx].result) {
        return COND_RESULT_FALSE;
    }

}

if (!con->conditional_is_valid[dc->comp]) {
    if (con->conf.log_condition_handling) {
      TRACE("cond[%d] is valid: %d", dc->comp,
            con->conditional_is_valid[dc->comp]);
    }

    return COND_RESULT_UNSET;
}
…
```

HUMAN STUDY TASKS

```
…
if (dc->prev) {
   if (con->conf.log_condition_handling) {
       log_error_write(srv, __FILE__, __LINE__,
      "sb", "go prev", dc->prev->key);
    }
    /* make sure prev is checked first */
    config_check_cond_cached(srv, con, dc->prev);
    /* one of prev set me to FALSE */
    if (COND_RESULT_FALSE == con->cond_cache[dc->context_ndx].result) {
        return COND_RESULT_FALSE;
    }

}

if (!con->conditional_is_valid[dc->comp]) {
    if (con->conf.log_condition_handling) {
      TRACE("cond[%d] is valid: %d", dc->comp,
            con->conditional_is_valid[dc->comp]);
    }

    return COND_RESULT_UNSET;
}
…
```

HUMAN STUDY TASKS

Question presentation

Question: What is the value of the variable "con->conditional_is_valid[dc->comp]" on line 35? (recall, you can use inequality symbols in your answer)

Answer to the Question Above:

HUMAN STUDY TASKS

```
…
if (dc->prev) {
   if (con->conf.log_condition_handling) {
       log_error_write(srv, __FILE__, __LINE__,
      "sb", "go prev", dc->prev->key);
    }
    /* make sure prev is checked first */
    config_check_cond_cached(srv, con, dc->prev);
    /* one of prev set me to FALSE */
    if (COND_RESULT_FALSE == con->cond_cache[dc->context_ndx].result) {
        return COND_RESULT_FALSE;
    }

}

if (!con->conditional_is_valid[dc->comp]) {
    if (con->conf.log_condition_handling) {
      TRACE("cond[%d] is valid: %d", dc->comp,
            con->conditional_is_valid[dc->comp]);
    }

    return COND_RESULT_UNSET;
}
…
```

HUMAN STUDY TASKS

Question presentation

Question: What is the value of the variable "con->conditional_is_valid[dc->comp]" on line 35? (recall, you can use inequality symbols in your answer)

Answer to the Question Above:

False

EVALUATION METRICS

Correctness – is the right answer reported?

Time – what is the "*maintenance effort*" associated with understanding this code?

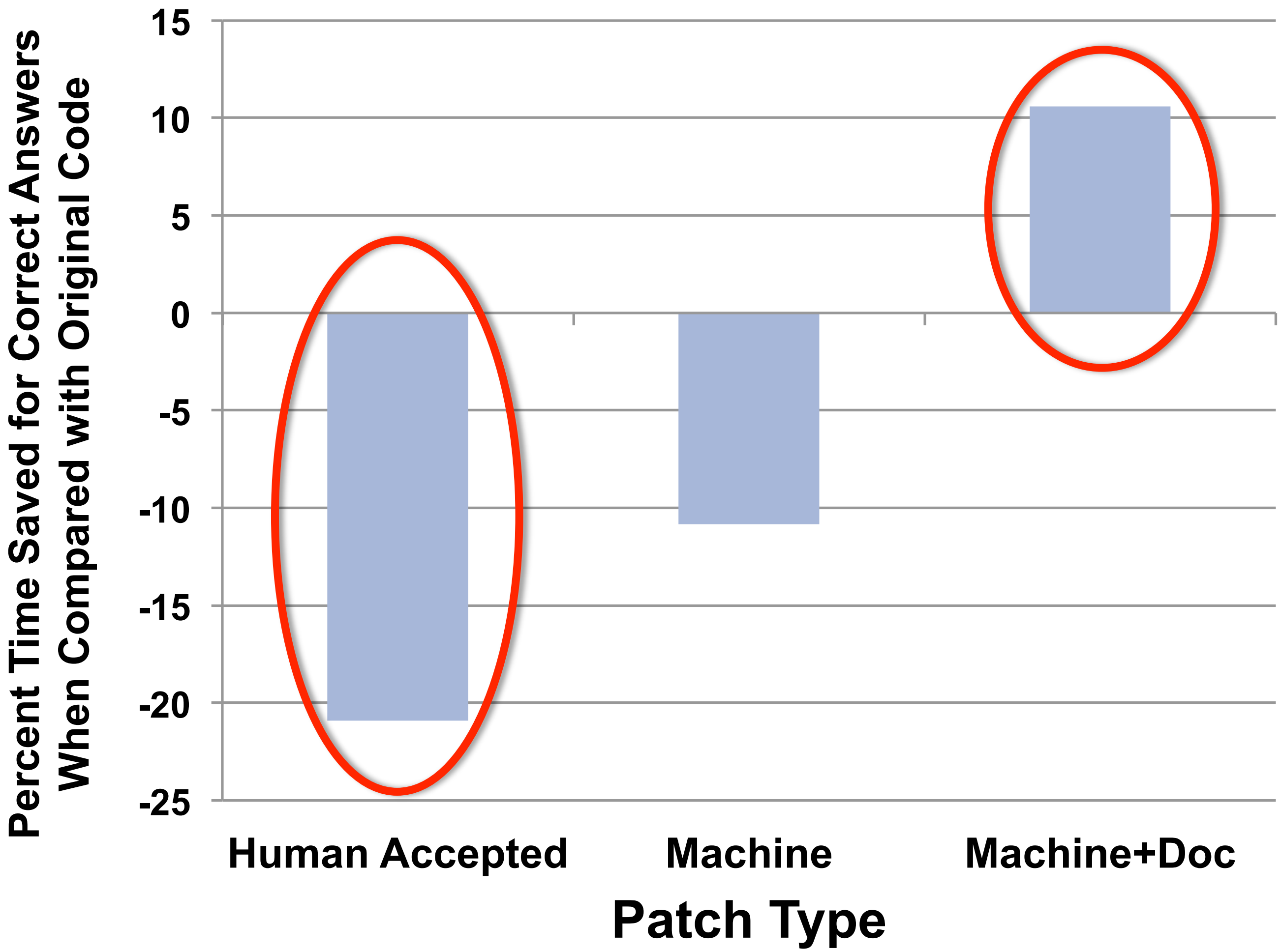
EVALUATION METRICS – RESULTS

Correctness – is the right answer reported?

Time – what is the "*maintenance effort*" associated with understanding this code?

- **Correctness was the same for all patches (with statistical significance)**
- **We then focus on time, as it represents the software engineering effort associated with program understanding**

TYPE OF PATCH VS. MAINTAINABILITY



Effort = average number of minutes it took participants to report a *correct* answer for all patches of a given type relative to the original code

CHARACTERISTICS OF MAINTAINABILITY

- **We measured various code features for all patches**
  - Using a logistic regression model, we can predict human accuracy 73.16% of the time
- **A Principle Component Analysis shows that 17 features are necessary to account for 90% of the variance in the data**
  - Modeling maintainability is a complex problem

CHARACTERISTICS OF MAINTAINABILITY

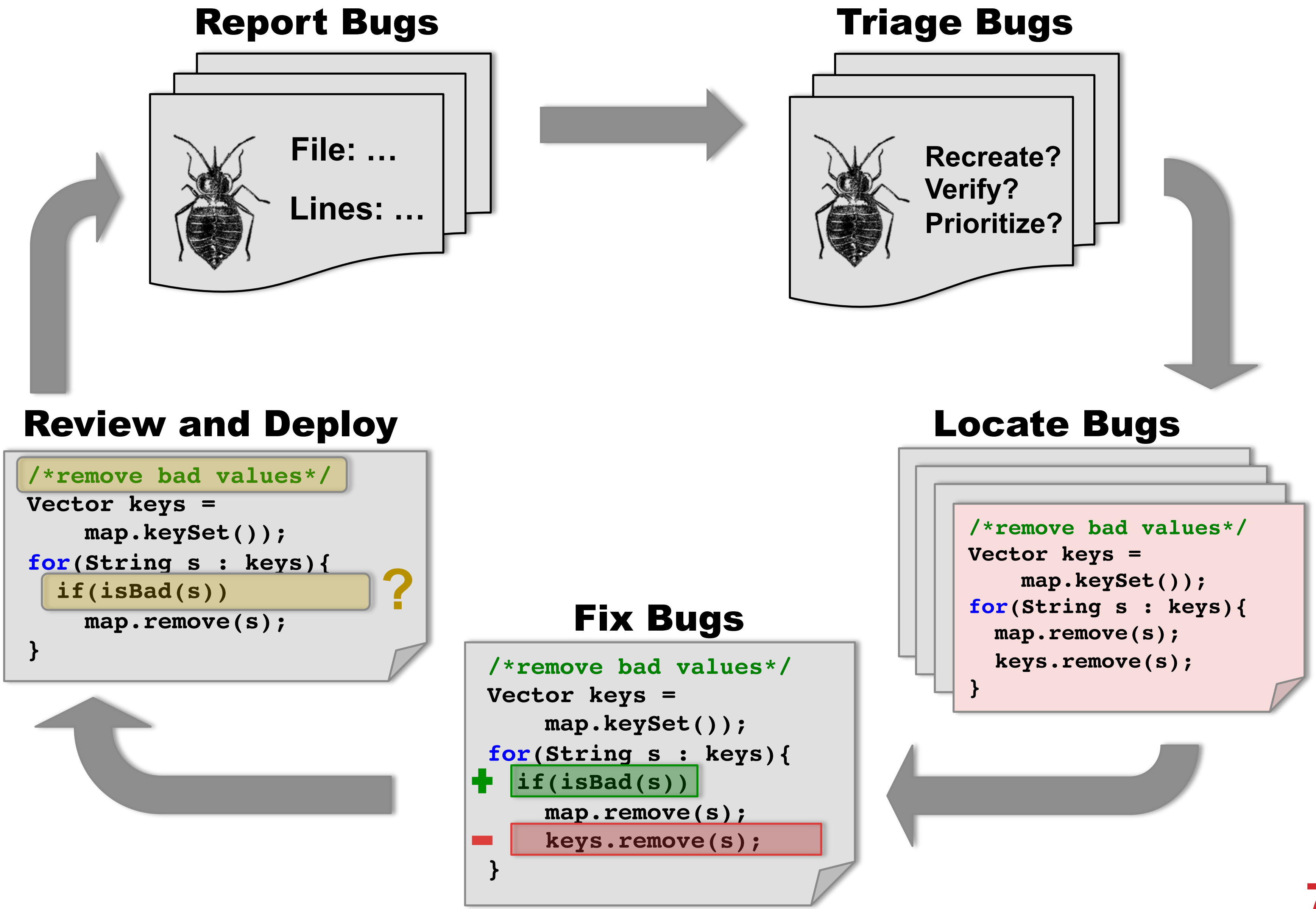
Code Feature	Predictive Power
Ratio of variable uses per assignment	0.178
Code readability	0.157
Ratio of variables declared out of scope vs. in scope	0.146
Number of total tokens	0.097
Number of non-whitespace characters	0.090
Number of macro uses	0.080
Average token length	0.078
Average line length	0.072
Number of conditionals	0.070
Number of variable declarations or assignments	0.056
Maximum conditional clauses on any path	0.055
Number of blank lines	0.054

HUMAN INTUITION VS. MEASUREMENT

After completing the study, participants were asked to report which code features they thought increased maintainability the most

Human Reported Feature	Votes	Predictive Power
Descriptive variable names	35	0.000
Clear whitespace and indentation	25	0.003
Presence of comments	25	0.022
Shorter function	8	0.000
Presence of nested conditionals	8	0.033
Presence of compiler directives / macros	7	0.080
Presence of global variables	5	0.146
Use of goto statements	5	0.000
Lack of conditional complexity	5	0.055
Uniform use and format of curly braces	5	0.014

MAINTENANCE OVERVIEW



- Westley Weimer, Zachary P. Fry, Stephanie Forrest, "Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results" Automated Software Engineering (ASE), 2013. (Acceptance rate: 23%)
- Zachary P. Fry, Westley Weimer, "Clustering Static Analysis Defect Reports to Reduce Maintenance Costs" Working Conference on Reverse Engineering (WCRE), 2013. (Acceptance rate: 39%)
- Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, Stephanie Forrest, "Software Mutational Robustness" Genetic Programming and Evolvable Machines, 2013.
- Zachary P. Fry, Bryan Landau, Westley Weimer, "A Human Study of Patch Maintainability" International Symposium on Software Testing and Analysis (ISSTA), 2012. (Acceptance rate: 29%)
- Zachary P. Fry, Westley Weimer, "Fault Localization Using Textual Similarities" Tech Report, Computing Research Repository, 2012.
- Zachary P. Fry, Westley Weimer, "A Human Study of Fault Localization Accuracy" International Conference of Software Maintenance (ICSM), 2010. (Acceptance rate: 26%)
- Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara, Yana Novikova, Lori L. Pollock, K. Vijay-Shanker, "AMAP: automatically mining abbreviation expansions in programs to enhance software maintenance tools," Working Conference on Mining Software Repositories (MSR), 2008. Best Paper Award (Acceptance rate: 40%)
- David Shepherd, Zachary P. Fry, Emily Gibson, Kishen Maloor, Lori Pollock, and K. Vijay- Shanker, "Introducing Natural Language Program Analysis (NLPA)", a research group presentation at the Workshop on Program Analysis for Software Tools and Engineering (PASTE), 2007.
- Zachary P. Fry, David Shepherd, Emily Hill, Lori Pollock, K. Vijay-Shanker, "Analyzing Source Code: Looking for Useful Verb-Direct Object Pairs in All the Right Places," The Institution of Engineering and Technology (IET) Software Special Issue on Natural Language in Software Development – Volume 2, Issue 1, 2007. (Impact Factor: 0.542)
- David Shepherd, Zachary P. Fry, Emily Gibson, Lori Pollock, and K. Vijay-Shanker, "Using Natural Language Program Analysis to Locate and Understand Action-Oriented Concerns," International Conference on Aspect Oriented Software Development (AOSD), 2007. (Acceptance rate: 18%)

COMPREHENSIVE GOALS - REVISITED

- **Generality**
  - Can cluster all attempted defect report types
  - AE can fix as many bug types as the state of the art tools
- **Usability**
  - Techniques work "*off the shelf*"
  - Ease incremental adoption
- **Comprehensive evaluation**
  - Humans agree with our defect report clusters
  - We find our patches with automated documentation are as maintainable as those created by humans

SUMMARY

Add lightweight analyses to specific tasks to reduce the overall cost of software maintenance

1. Reducing triage/fix costs by clustering defect reports
2. Speeding up an automatic patch generation technique
3. Exploring the maintainability of various types of patches