Search-Based Software Engineering:

Foundations and Recent Applications

Ali Ouni

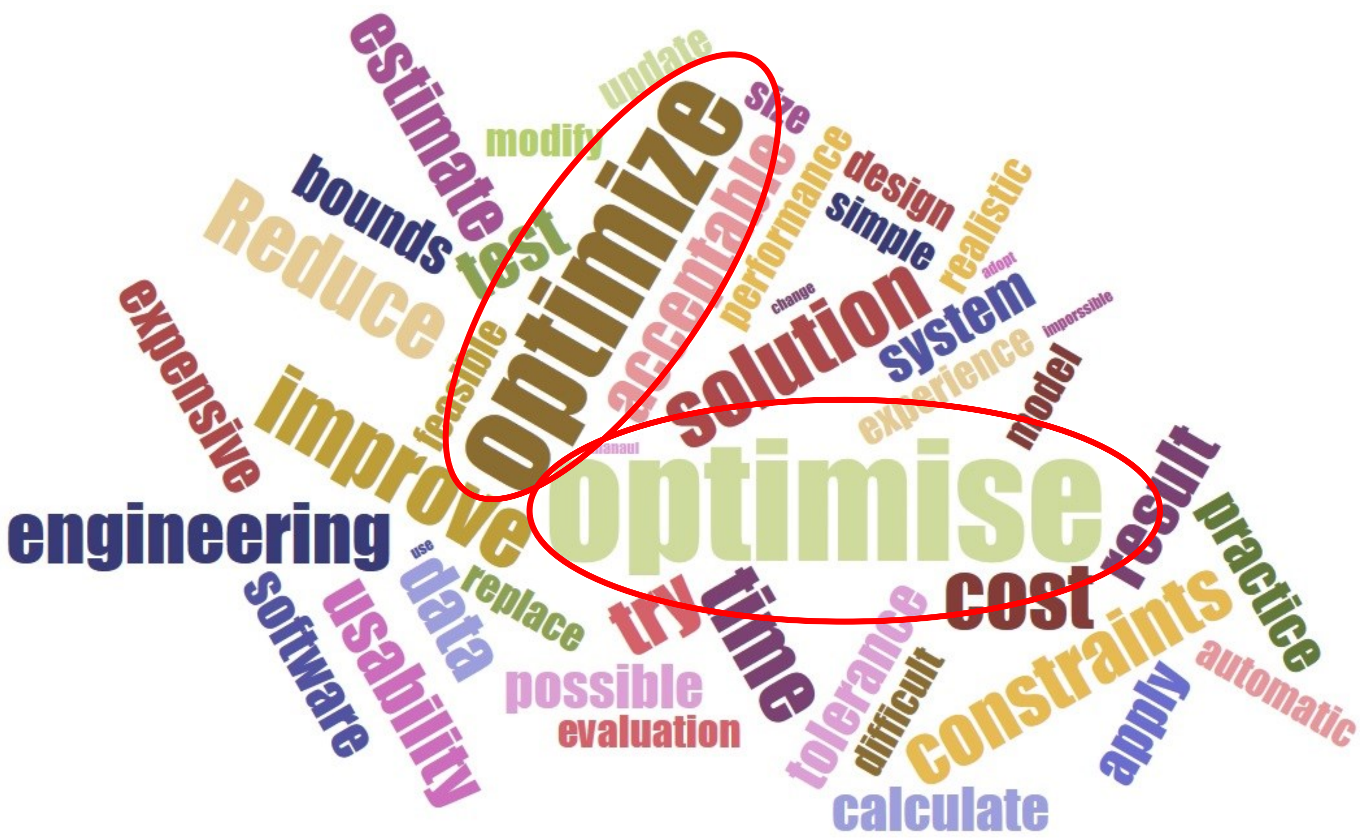
Software Engineering Lab, Osaka University, Japan

5th Asian Workshop of Advanced Software Engineering (AWASE'16), 19-20 March, 2016, Nara, Japan

Engineering Words

estimate
update
size
modify
optimize
bounds
Reduce
test
acceptable
performance
design
simple
realistic
adopt
change
solution
system
imporssible
expensive
feasible
experience
model
improve
manaul
optimise
result
engineering
use
practice
data
replace
try
time
cost
software
usability
tolerance
constraints
automatic
possible
difficult
apply
evaluation
calculate

Engineering Words



Optimi**s**e
Optimi**z**e

so good they named it twice!

➡ SBSE

What is SBSE ?

The term “Search-Based Software Engineering” (SBSE) coined in 2001 by Mark Harman.

- SBSE uses intelligent search techniques to explore large search spaces, guided by a fitness function that captures properties of the desirable solutions we seek.

Genetic Programming

Ant Colonies

Harmony Search

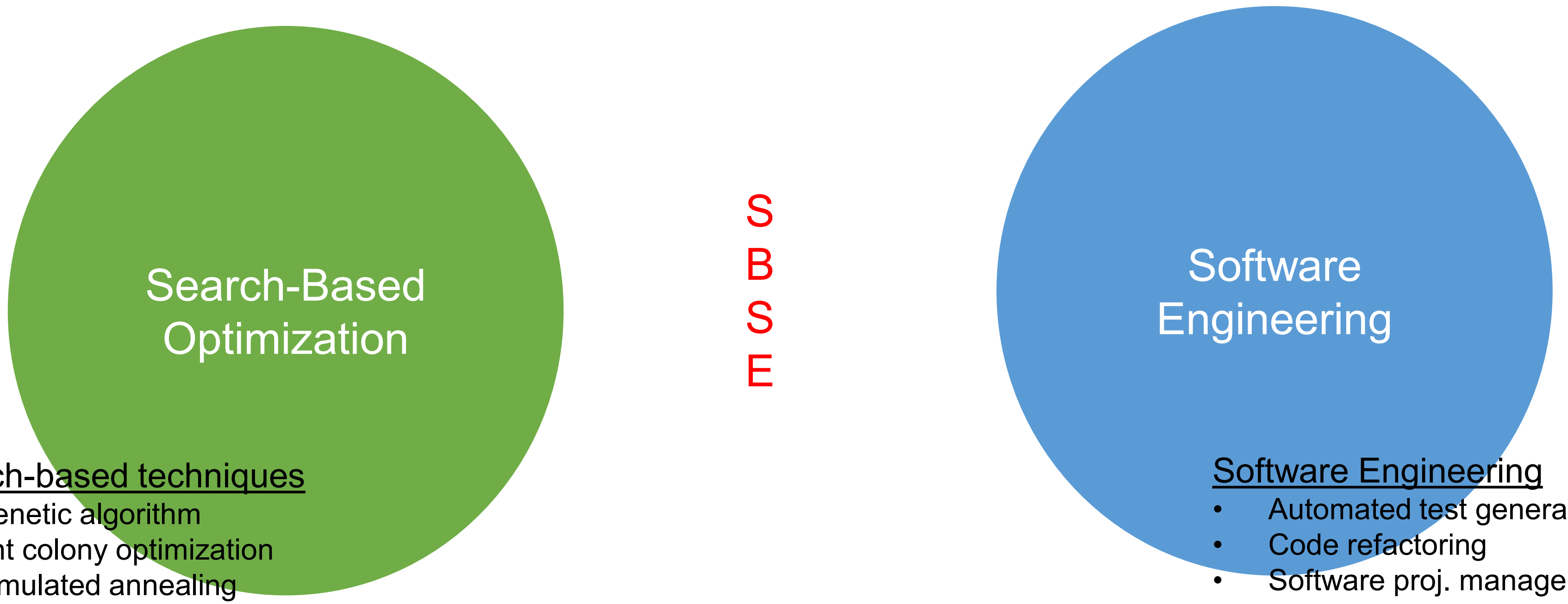
Hill Climbing

Particle Swarm Optimization

Tabu Search

Simulated Annealing

What is SBSE ?



Search-Based Optimization

S
B
S
E

Software Engineering

Search-based techniques

- Genetic algorithm
- Ant colony optimization
- Simulated annealing
- Particle swarm optimization
- NSGA-II
- NSGA-III
- . . .

Software Engineering

- Automated test generation
- Code refactoring
- Software proj. management
- Requirements engineering
- Fault localization
- Program repair
- . . .

SBSE

- Search-based Testing
- Search-based refactoring
- Search-based software management
- Search-based requirements engineering
- Search-based Fault Location
- Search-based program repair
- . . .

Conventional vs Search-based

- Conventional Software Engineering
  - Write a method to construct a good solution

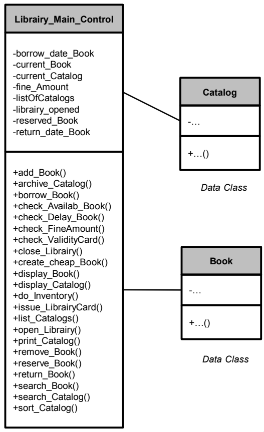
- Search Based Software Engineering
  - Write a method to guide you to the best solution

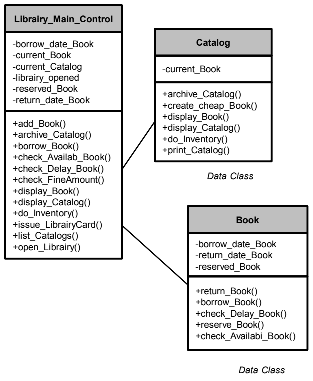
SBSE for software refactoring

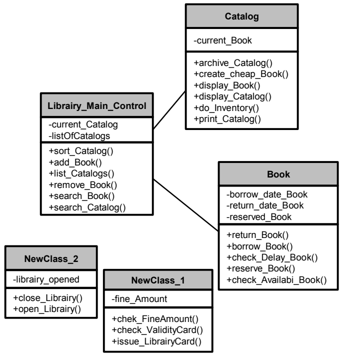
- SBSE formulation
  1. Identify a software engineering problem
  2. Identify the desirable properties of a good solution you would like to have
  3. Formulate them in a measurable way
  4. Use them as a way for searching the space of possible solutions

Optimize:

1. Coupling
2. Cohesion



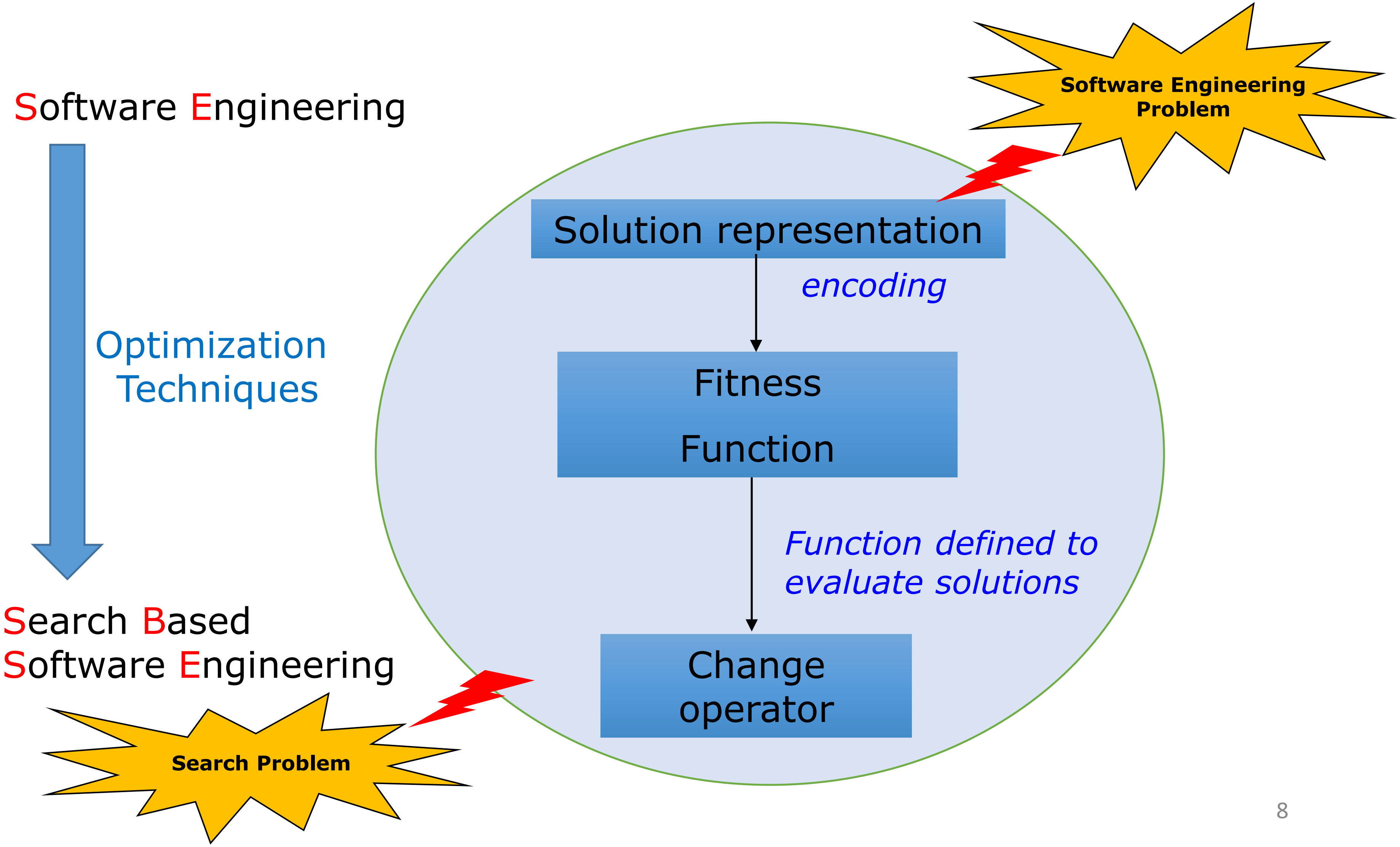




Next ?

$$Coupling(MS) = \sum_{i=1}^{n} I(S_i) - I(S)$$

SBSE in a nutshell …



Software Engineering

Optimization Techniques

Search Based Software Engineering

Search Problem

Software Engineering Problem

Solution representation

encoding

Fitness Function

Function defined to evaluate solutions

Change operator

Is SBSE interesting?

let’s listen to software engineers ...

... what sort of things do they say?

Software Engineers Say…

Requirements: We need all requirements that balance software development cost and customer satisfaction

Management: We need to reduce risk while maintaining completion time

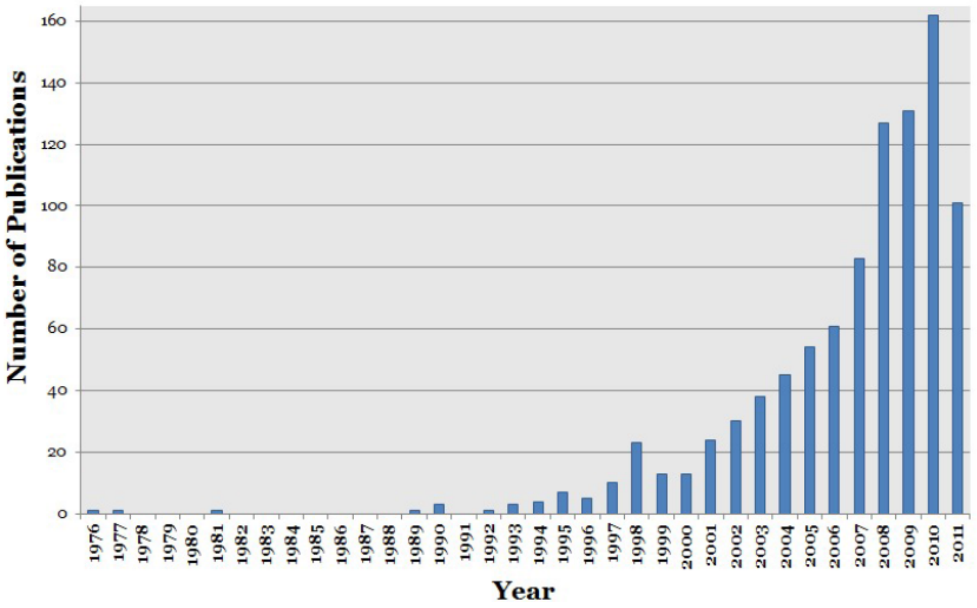
Design: We need increased cohesion and decreased coupling

Testing: We need fewer tests that find more nasty bugs

Refactoring: We need to optimize for all metrics M1,...., Mn

All have been addressed in the SBSE literature!

… but …
why is SBSE growing very fast?



Publication growth up to 2012

- 1,600 authors
- nearly 300 institutions
- more than 40 countries

- TOP conferences in SE
  - ICSE and FSE : whole sessions to SBSE
- TOP conferences in Evolutionary computation
  - GECCO: have track dedicated to SBSE
- Dedicated international conferences: (SSBSE, SBST) and many other workshops

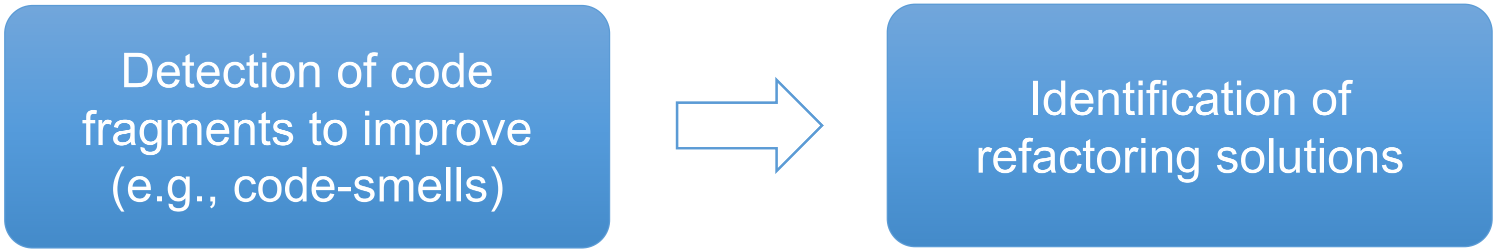
Search-based refactoring

- Systems, like people, get old : increase in complexity and degrade in effectiveness
- Software changes frequently
  - Add new requirements
  - Adapt to environment changes
  - Correct bugs
- Changing a software can be a challenging task
  - These changes may degrade their design and QoS
  - The original developers are not around anymore
- Maintain a high level of quality during the life cycle of a software system

Software refactoring

- The process of improving a code after it has been written by changing its internal structure without changing the external behavior (Fowler et al., ‘99)
  - Examples: *Move method, extract class, move attribute, ...*
  - *IDEs: Eclipse, NetBeans, …*

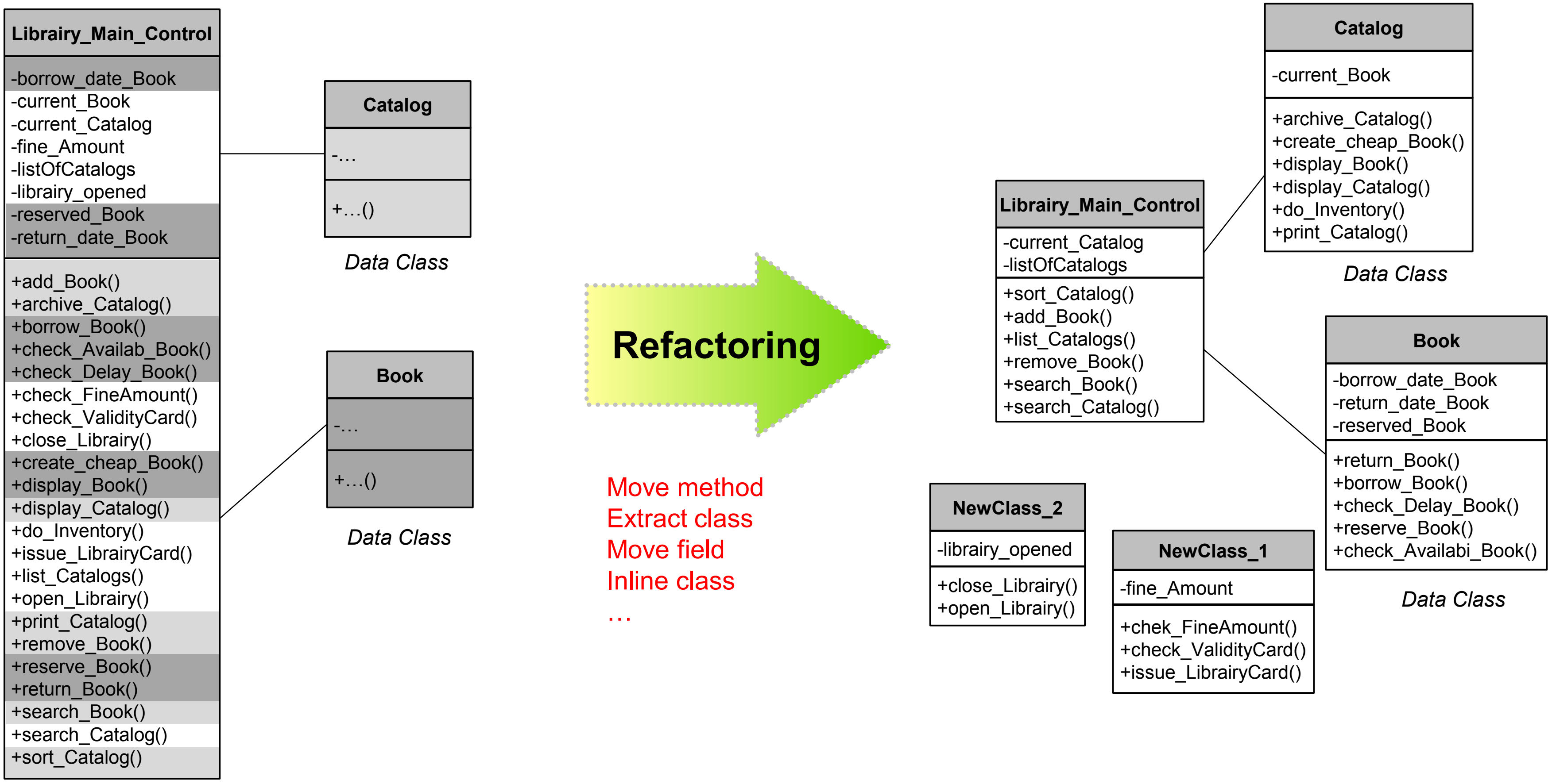
- Two steps



Detection of code fragments to improve (e.g., code-smells)

Identification of refactoring solutions

Step 2: Refactoring



Librairy_Main_Control

-borrow_date_Book
-current_Book
-current_Catalog
-fine_Amount
-listOfCatalogs
-librairy_opened
-reserved_Book
-return_date_Book

+add_Book()
+archive_Catalog()
+borrow_Book()
+check_Availab_Book()
+check_Delay_Book()
+check_FineAmount()
+check_ValidityCard()
+close_Librairy()
+create_cheap_Book()
+display_Book()
+display_Catalog()
+do_Inventory()
+issue_LibrairyCard()
+list_Catalogs()
+open_Librairy()
+print_Catalog()
+remove_Book()
+reserve_Book()
+return_Book()
+search_Book()
+search_Catalog()
+sort_Catalog()

Catalog

-…

+…()

Data Class

Book

-…

+…()

Data Class

Refactoring

Move method
Extract class
Move field
Inline class
…

Catalog

-current_Book

+archive_Catalog()
+create_cheap_Book()
+display_Book()
+display_Catalog()
+do_Inventory()
+print_Catalog()

Data Class

Librairy_Main_Control

-current_Catalog
-listOfCatalogs

+sort_Catalog()
+add_Book()
+list_Catalogs()
+remove_Book()
+search_Book()
+search_Catalog()

Book

-borrow_date_Book
-return_date_Book
-reserved_Book

+return_Book()
+borrow_Book()
+check_Delay_Book()
+reserve_Book()
+check_Availabi_Book()

Data Class

NewClass_2

-librairy_opened

+close_Librairy()
+open_Librairy()

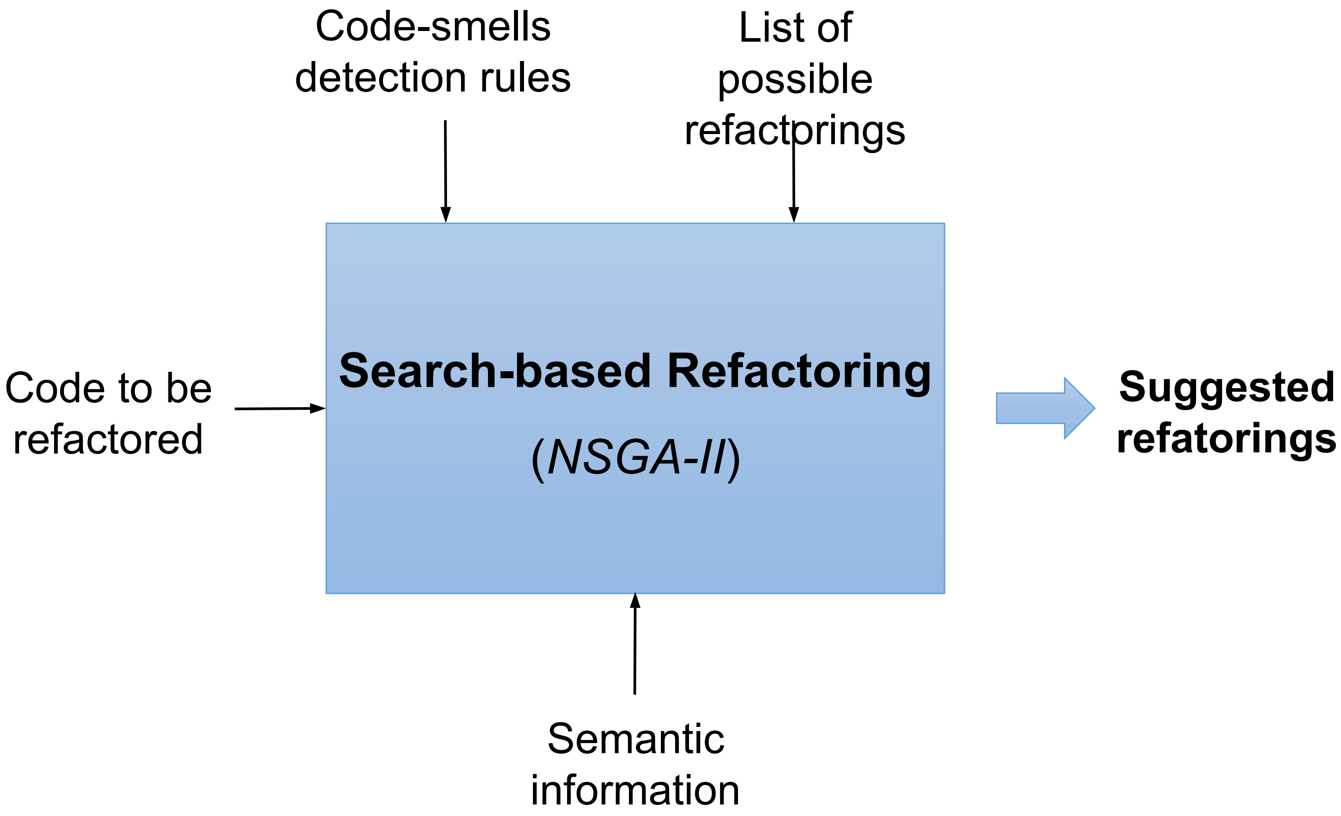
NewClass_1

-fine_Amount

+chek_FineAmount()
+check_ValidityCard()
+issue_LibrairyCard()

Blob

Search-based Refactoring



Code-smells detection rules

List of possible refactorings

Code to be refactored

Search-based Refactoring

(NSGA-II)

Suggested refatorings

Semantic information

Ali Ouni, Marouane Kessentini, Houari Sahraoui, Katsuro Inoue, Kalyanmoy Deb, "Multi-criteria Code Refactoring Suggestions: An Industrial Case Study", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2016.

NSGA-II adaptation

- Solution representation
  - Individual = Sequence of refactoring operations

1	moveMethod
2	pullUpAttribute
3	extractClass
4	inlineClass
5	extractSuperClass
6	inlineMethod

  - Controlling parameters

Refactorings	Controlling parameters
move method	(sourceClass, targetClass, method)
move field	(sourceClass, targetClass, field)
pull up field	(sourceClass, targetClass, field)
pull up method	(sourceClass, targetClass, method)
push down field	(sourceClass, targetClass, field)
push down method	(sourceClass, targetClass, method)
inline class	(sourceClass, targetClass)
extract class	(sourceClass, newClass)

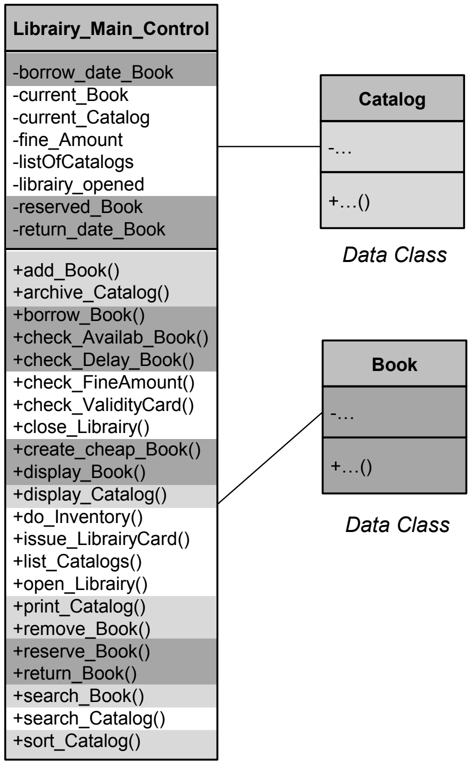
NSGA-II adaptation

- Four refactoring objectives

  1. Quality
     - Minimize the number of code-smells
  2. Code changes
     - Minimize the number of code changes required when applying refactoring
  3. Preserve semantic coherence
     - Maximize semantics coherence of the refactored code

Refactoring objectives

- Number of code changes



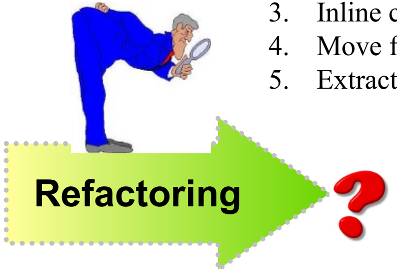
Code-smell: Blob

Solution 1

1. Move method
2. Extract class
3. Move field
4. Move method
5. Move method

Solution 2

1. Move method.
2. Move method
3. Inline class
4. Move field
5. Extract class

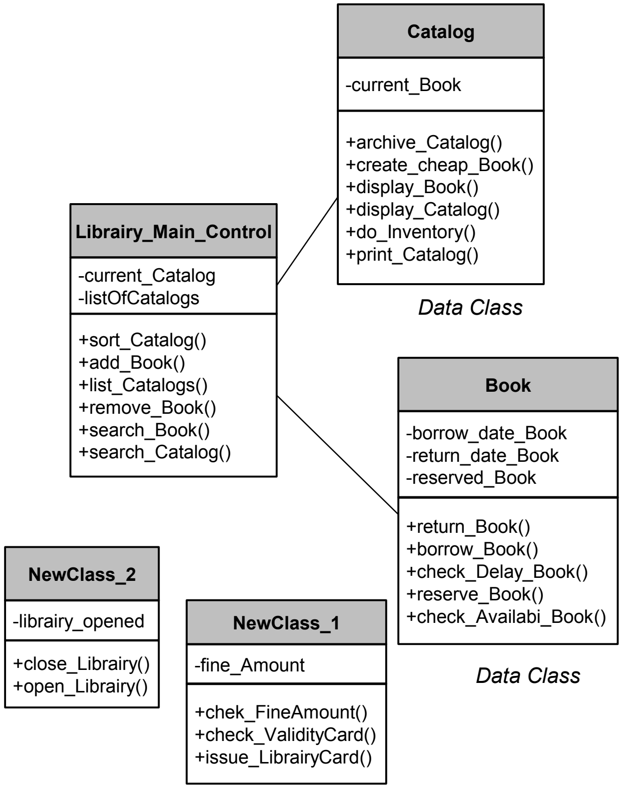


Solution 3

1. Move method
2. Extract class
3. Move field
4. Move method
5. Move method
6. Inline class
7. Move field
8. Extract class
9. Move method
10 Move field

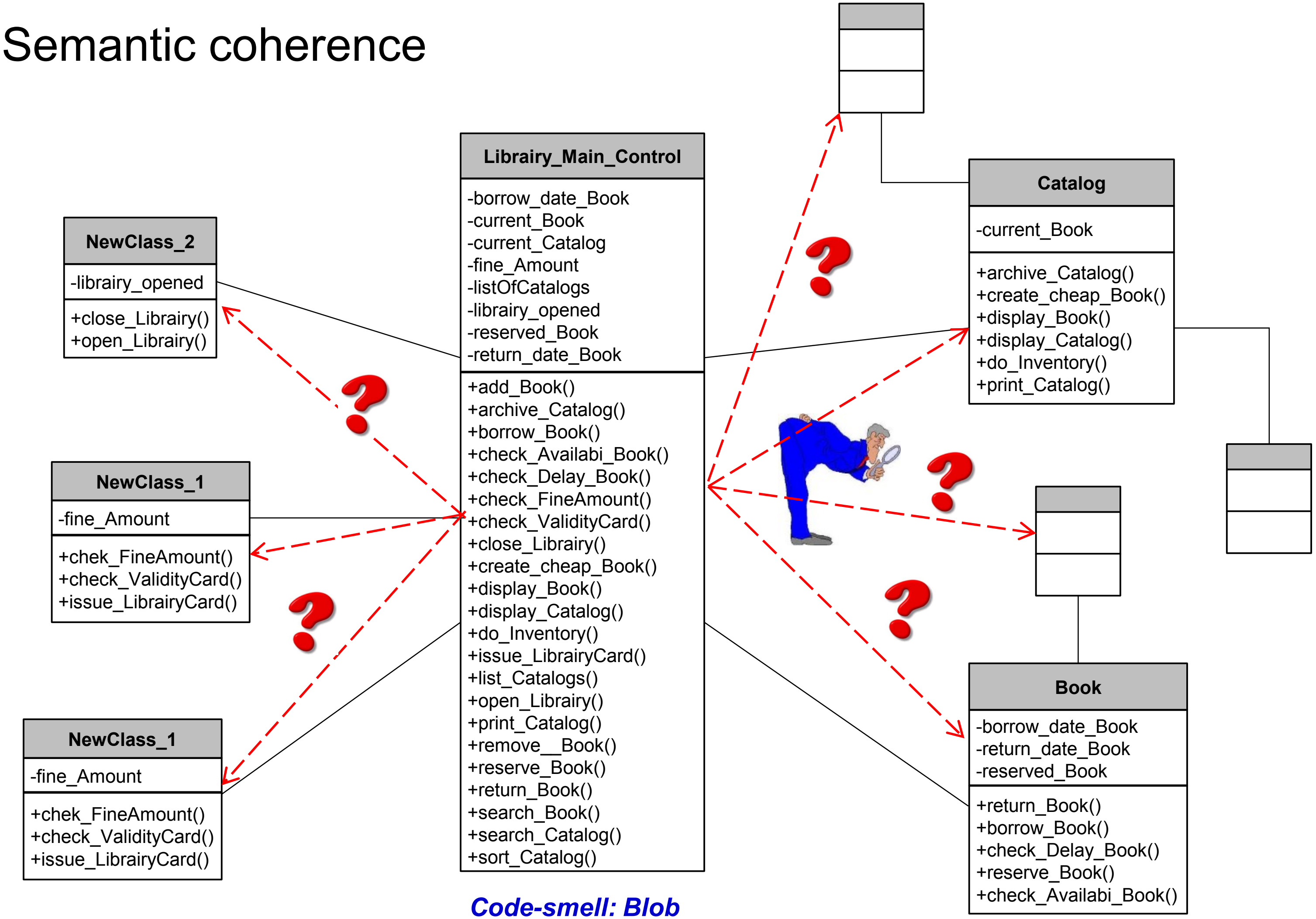
Solution 4

1. Move method.
2. Move method
3. Inline class
4. Move field
5. Extract class
6. Move field
7. Extract class



Refactoring objectives

- Semantic coherence



Librairy_Main_Control

-borrow_date_Book
-current_Book
-current_Catalog
-fine_Amount
-listOfCatalogs
-librairy_opened
-reserved_Book
-return_date_Book

+add_Book()
+archive_Catalog()
+borrow_Book()
+check_Availabi_Book()
+check_Delay_Book()
+check_FineAmount()
+check_ValidityCard()
+close_Librairy()
+create_cheap_Book()
+display_Book()
+display_Catalog()
+do_Inventory()
+issue_LibrairyCard()
+list_Catalogs()
+open_Librairy()
+print_Catalog()
+remove__Book()
+reserve_Book()
+return_Book()
+search_Book()
+search_Catalog()
+sort_Catalog()

NewClass_2

-librairy_opened

+close_Librairy()
+open_Librairy()

NewClass_1

-fine_Amount

+chek_FineAmount()
+check_ValidityCard()
+issue_LibrairyCard()

NewClass_1

-fine_Amount

+chek_FineAmount()
+check_ValidityCard()
+issue_LibrairyCard()

Catalog

-current_Book

+archive_Catalog()
+create_cheap_Book()
+display_Book()
+display_Catalog()
+do_Inventory()
+print_Catalog()

Book

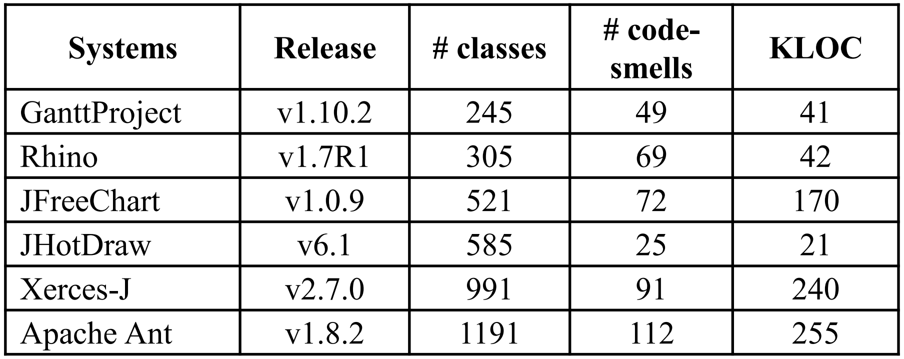
-borrow_date_Book
-return_date_Book
-reserved_Book

+return_Book()
+borrow_Book()
+check_Delay_Book()
+reserve_Book()
+check_Availabi_Book()

Code-smell: Blob

Evaluation

- Studied systems



Systems	Release	# classes	# code-smells	KLOC
GanttProject	v1.10.2	245	49	41
Rhino	v1.7R1	305	69	42
JFreeChart	v1.0.9	521	72	170
JHotDraw	v6.1	585	25	21
Xerces-J	v2.7.0	991	91	240
Apache Ant	v1.8.2	1191	112	255

- Three code-smell types
  - Blob
  - Spaghetti code
  - Functional decomposition

Empirical evaluation

- Quantitative
  - Number of fixed code smells
  - Number of code changes
  - QMOOD: Effectiveness, Flexibility, Reusability, Understandability

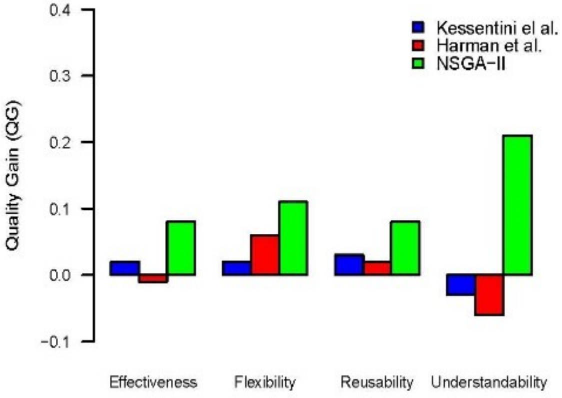
- Qualitative
  - Survey: evaluate the "relevance" of the suggested refactorings
  - Sample of 10 refactoring operations
  - 18 subjects

- Comparison to state-of-the-art research
  - Harman et al 2007, Basic GA approach

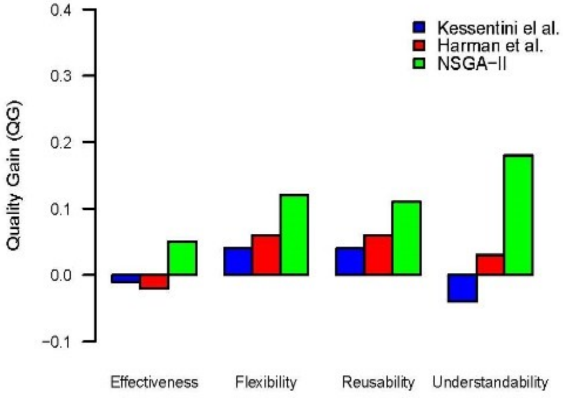
Refactoring results

<table>
<tr><th>Systems</th><th>Approach</th><th>Smells correction</th><th>Semantic coherence</th><th>Changes score</th></tr>
<tr><td rowspan="3">Xerces</td><td>NSGA-II</td><td>83% (55|66)</td><td>81 %</td><td>3843</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>41 %</td><td>2669</td></tr>
<tr><td>GA-based approach</td><td>89% (59/66)</td><td>37 %</td><td>4998</td></tr>
<tr><td rowspan="3">JFreeChart</td><td>NSGA-II</td><td>86% (49|57)</td><td>82 %</td><td>2016</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>36 %</td><td>3269</td></tr>
<tr><td>GA-based approach</td><td>91% (52\57)</td><td>37 %</td><td>3389</td></tr>
<tr><td rowspan="3">GanttProject</td><td>NSGA-II</td><td>85% (35|41)</td><td>80 %</td><td>2826</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>23 %</td><td>4790</td></tr>
<tr><td>GA-based approach</td><td>95% (39|41)</td><td>27 %</td><td>4697</td></tr>
<tr><td rowspan="3">AntApache</td><td>NSGA-II</td><td>78% (64|82)</td><td>78 %</td><td>4690</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>40 %</td><td>6987</td></tr>
<tr><td>GA-based approach</td><td>80% (66|82)</td><td>30 %</td><td>6797</td></tr>
<tr><td rowspan="3">JHotDraw</td><td>NSGA-II</td><td>86% (18|21)</td><td>80 %</td><td>2231</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>37 %</td><td>3654</td></tr>
<tr><td>GA-based approach</td><td>% (|21)</td><td>43 %</td><td>3875</td></tr>
<tr><td rowspan="3">Rhino</td><td>NSGA-II</td><td>85% (52|61)</td><td>80 %</td><td>1914</td></tr>
<tr><td>Harman et al. '07</td><td>N.A</td><td>37 %</td><td>2698</td></tr>
<tr><td>GA-based approach</td><td>87% (53|61)</td><td>32 %</td><td>3365</td></tr>
<tr><td rowspan="3"><b>Average
(<i>all systems</i>)</b></td><td><b>NSGA-II</b></td><td><b>84%</b></td><td><b>80 %</b></td><td><b>2937</b></td></tr>
<tr><td><b>Harman et al. '07</b></td><td><b>N.A</b></td><td><b>36 %</b></td><td><b>4011</b></td></tr>
<tr><td><b>GA-based approach</b></td><td><b>89%</b></td><td><b>34 %</b></td><td><b>4520</b></td></tr>
</table>

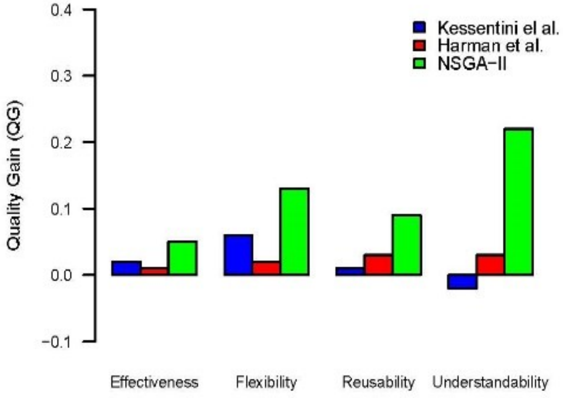
Refactoring results



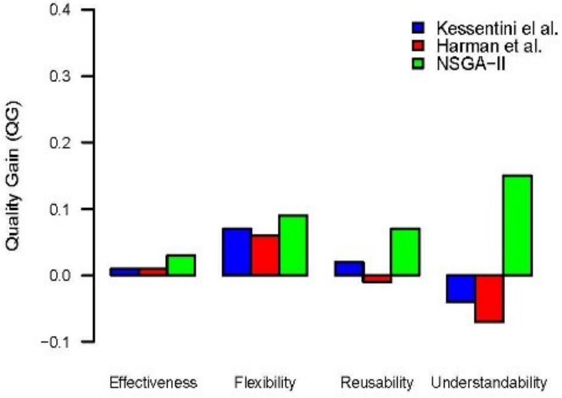
(a) Xerces-J



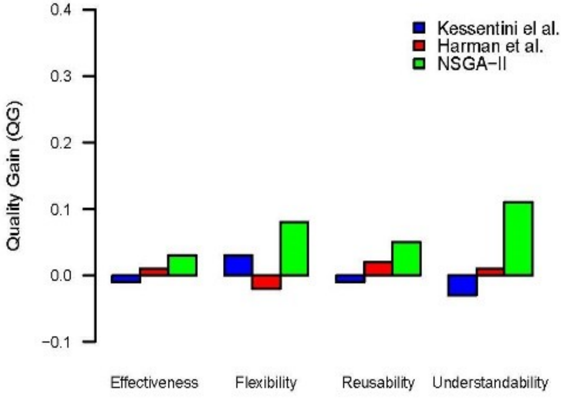
(b) JFreeChart



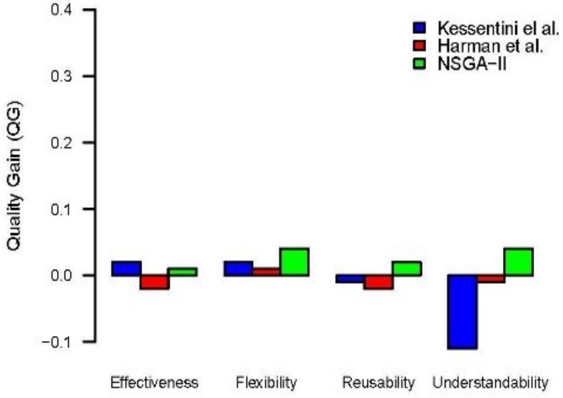
(c) GanttProject



(d) AntApache

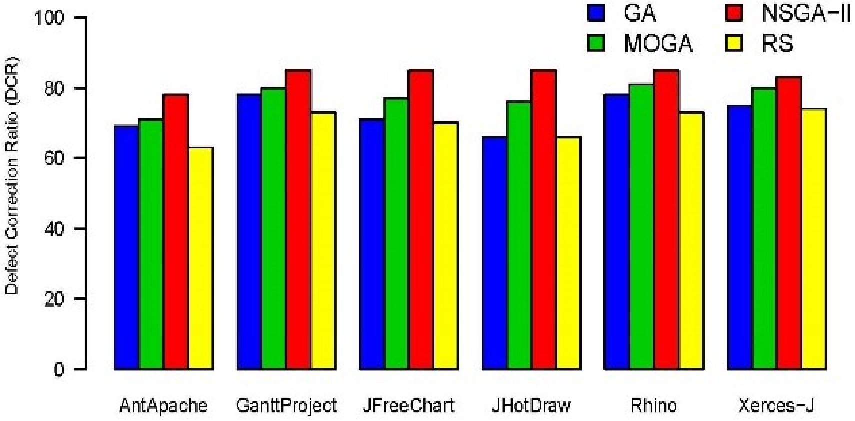


(e) Rhino

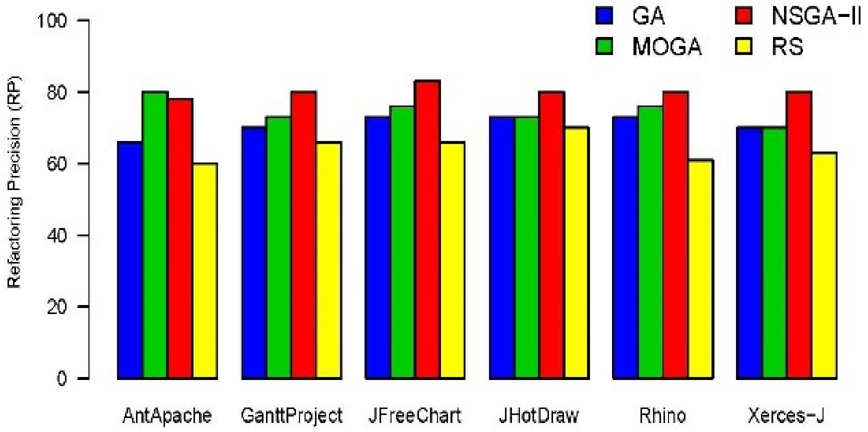


(f) JHotDraw

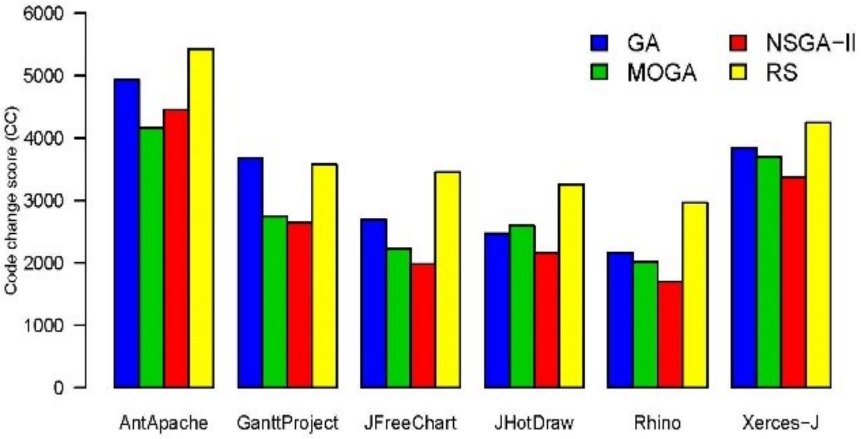
SBSE Validation



(a) Defect Correction Ratio

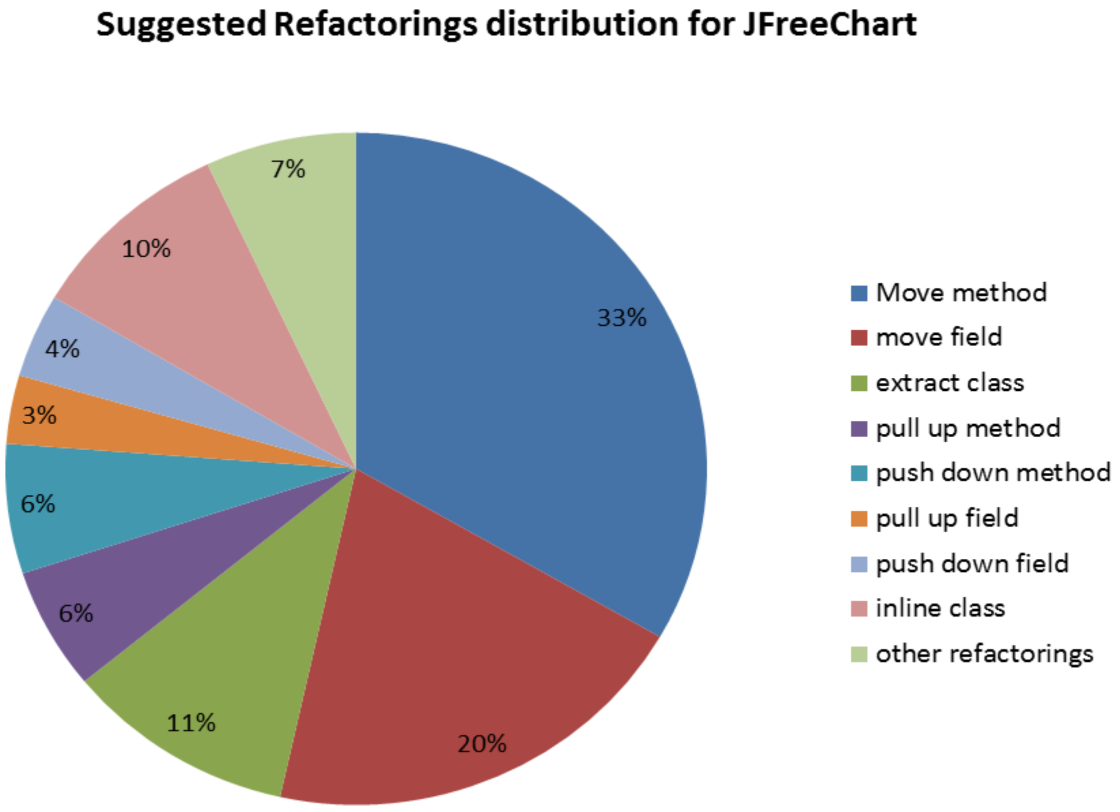


(b) Refactoring Precision



(c) Code Change Score

Refactoring distribution



Conclusion

- SBSE: write a fitness function to guide automated search
- SBSE formulation
  1. Identify the desirable properties of a good solution you would like to have
  2. Formulate them in a measurable way
  3. Use them as a way for searching the space of possible solutions

- SBSE is applied to solve problems in all software lifecycle
  - Requirements engineering
  - Software project management
  - Design
  - Refactoring
  - Software testing

- Provides scalable, realistic, robust and generic solutions

Thank You!

Questions?