Université de Montréal

Department of Computer Science and Operations Research

A mono- and Multi-objective Approach for Recommending Software Refactoring

Ali Ouni

Ph.D. Defense

Advisors: Houari Sahraoui (Université de Montréal, Canada)
Marouane Kessentini (University of Michigan, USA)

SBSE

12 November 2014

geodes

Outline

- Context and problem
- Research methodology
- Code-smells detection
- Mono-objective software refactoring
- Multi-objective software refactoring
- Conclusion and perspectives

Context

- Software systems have become prevalent in our everyday life
- 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
- Easiness to accommodate changes depends on software quality
- Maintain a high level of quality during the life cycle of a software system

Refactoring

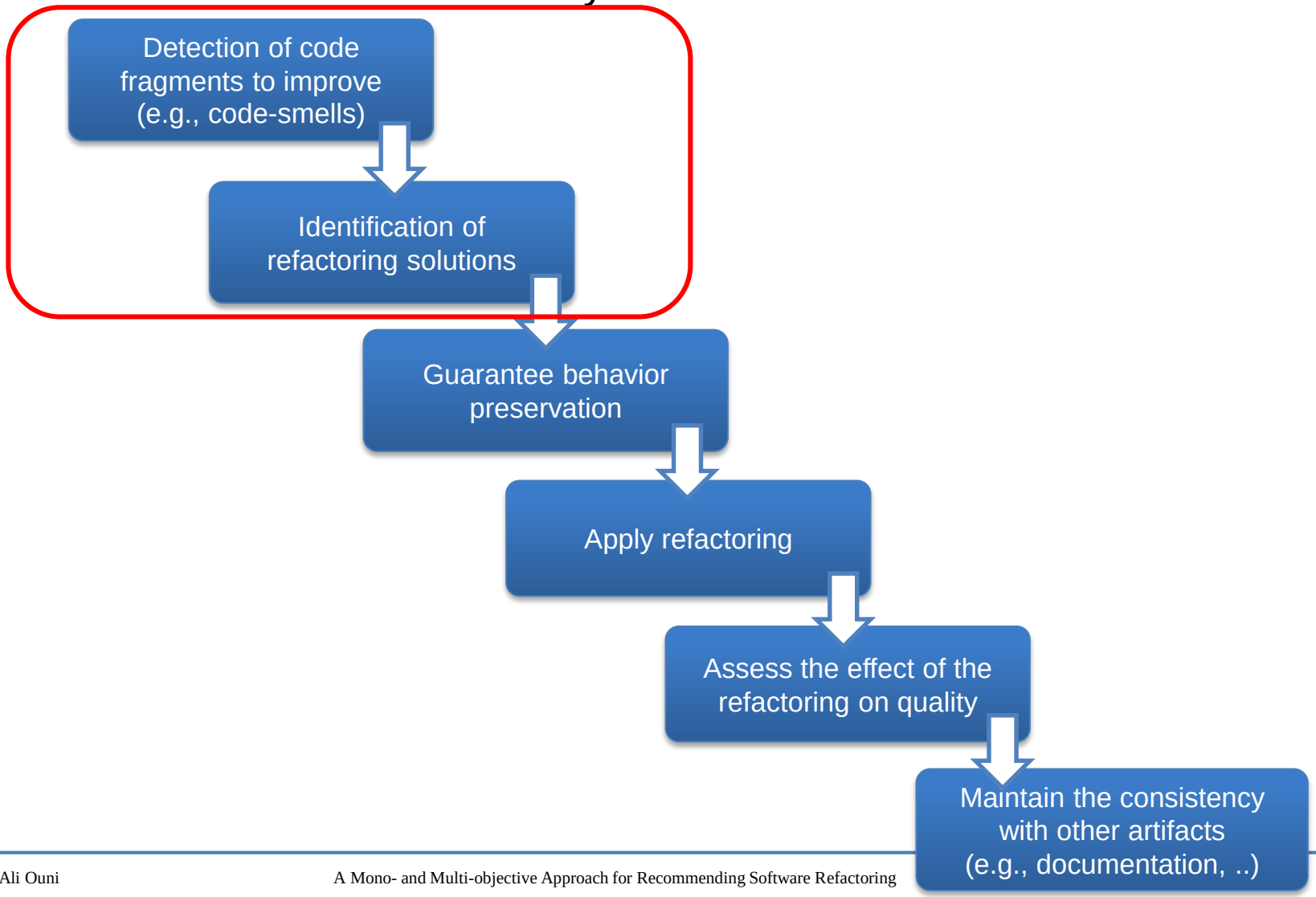
- " The process of improving a code after it has been written by changing its internal structure without changing the external behavior " (Fowler et al., '99)
  - Examples: *Move method*, *extract class*, *move attribute*, ...
  - Eclipse, NetBeans, …

- Advantages
  - Improve software quality, maintainability, readability
  - Provide better software extensibility
  - Increase the speed at which programmers can write and maintain their code

- Challenges…
  - Manual refactoring is an error-prone task
  - What are the situations ? What are the refactorings to apply?

Refactoring

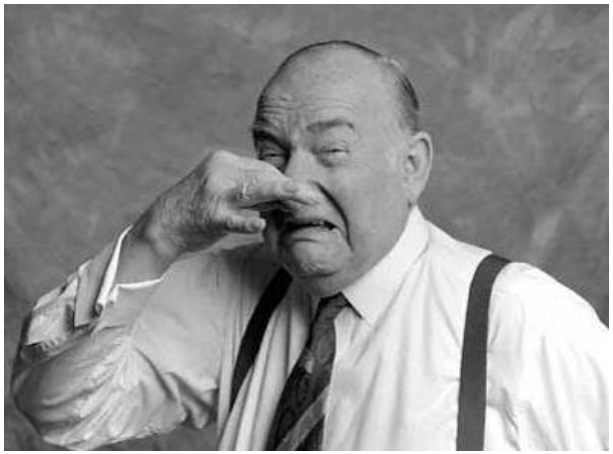
- Need for recommendation systems



Step 1: Code-smells detection

- Code-smells
  - Introduced during the initial design or during evolution
  - Anomalies, anti-patterns, bad smells, design flaws, …

- " Metaphor " to describe problems resulting from bad design and programming practices
  - Time pressure, non-experienced programmers, unintentionally, code decay
  - Lead to software products suffering by poor performance and QoS
  - Code difficult to understand, modify, maintain, evolve ...



Code smells

What is that smell? Did you write that code?

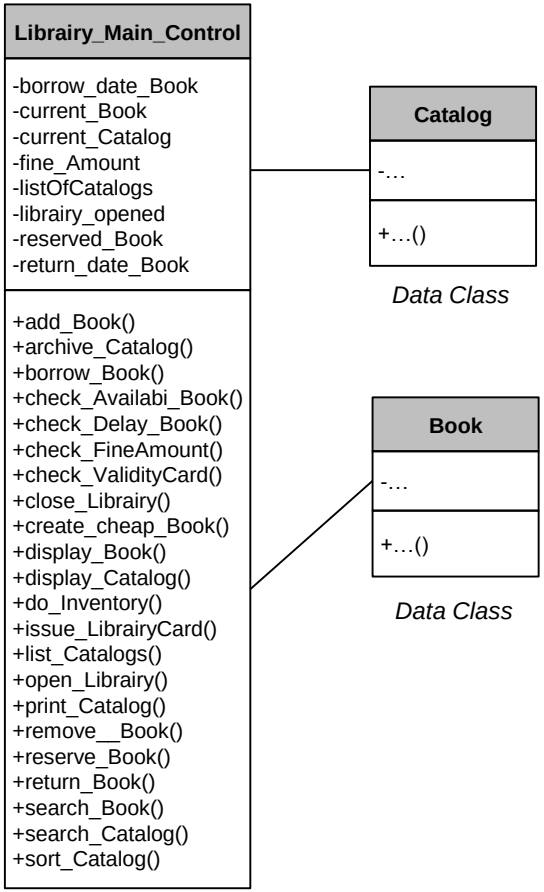
Code-smell examples

Spaghetti code

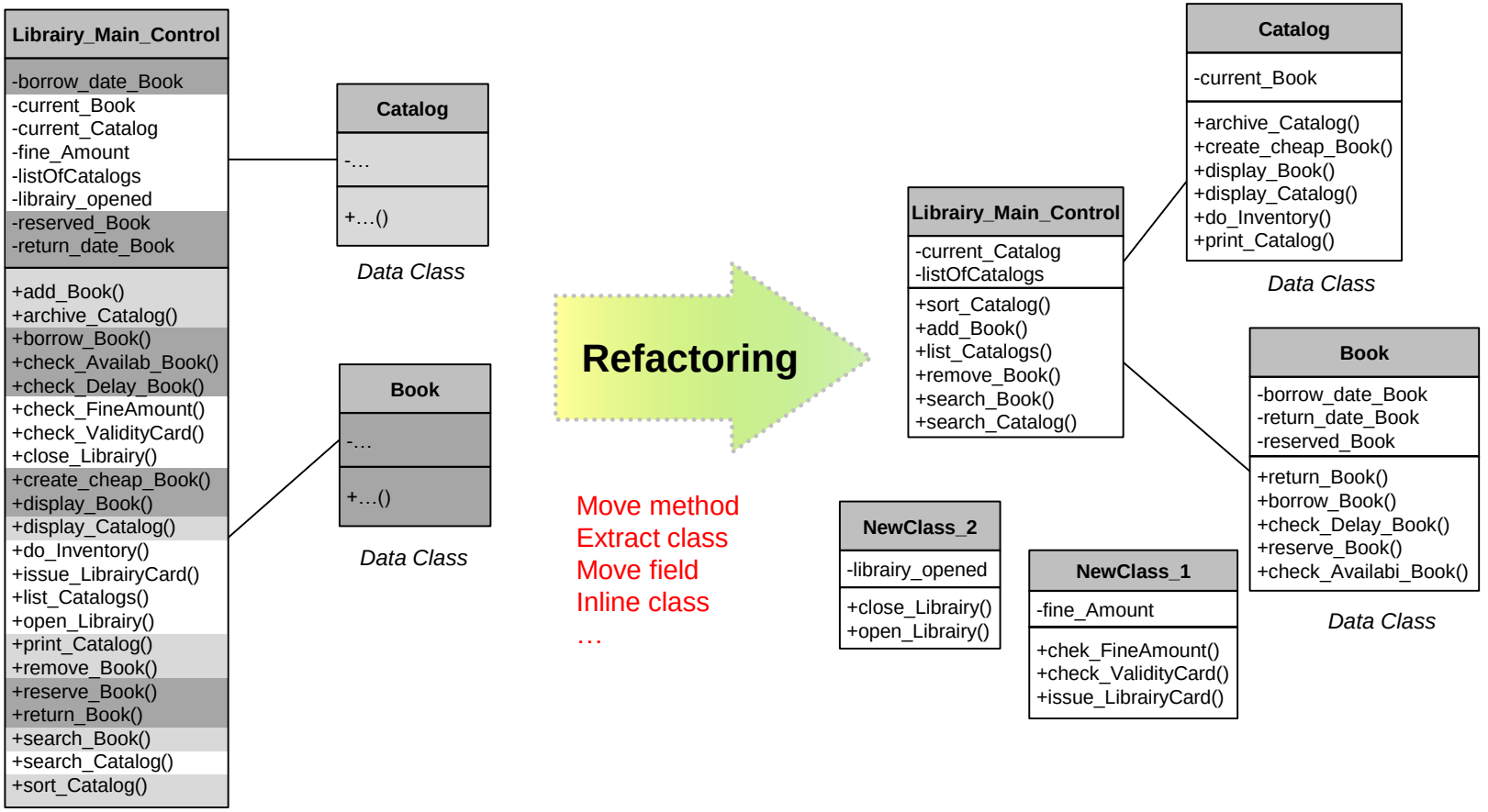
"It is a code with a complex and tangled control structure. This code-smell is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes with no structure, declaring long methods with no parameters, and utilising global variables"

Blob

"Procedural-style design leads to one object with numerous responsibilities and most other objects only holding data or executing simple processes"



Step 2: Refactoring



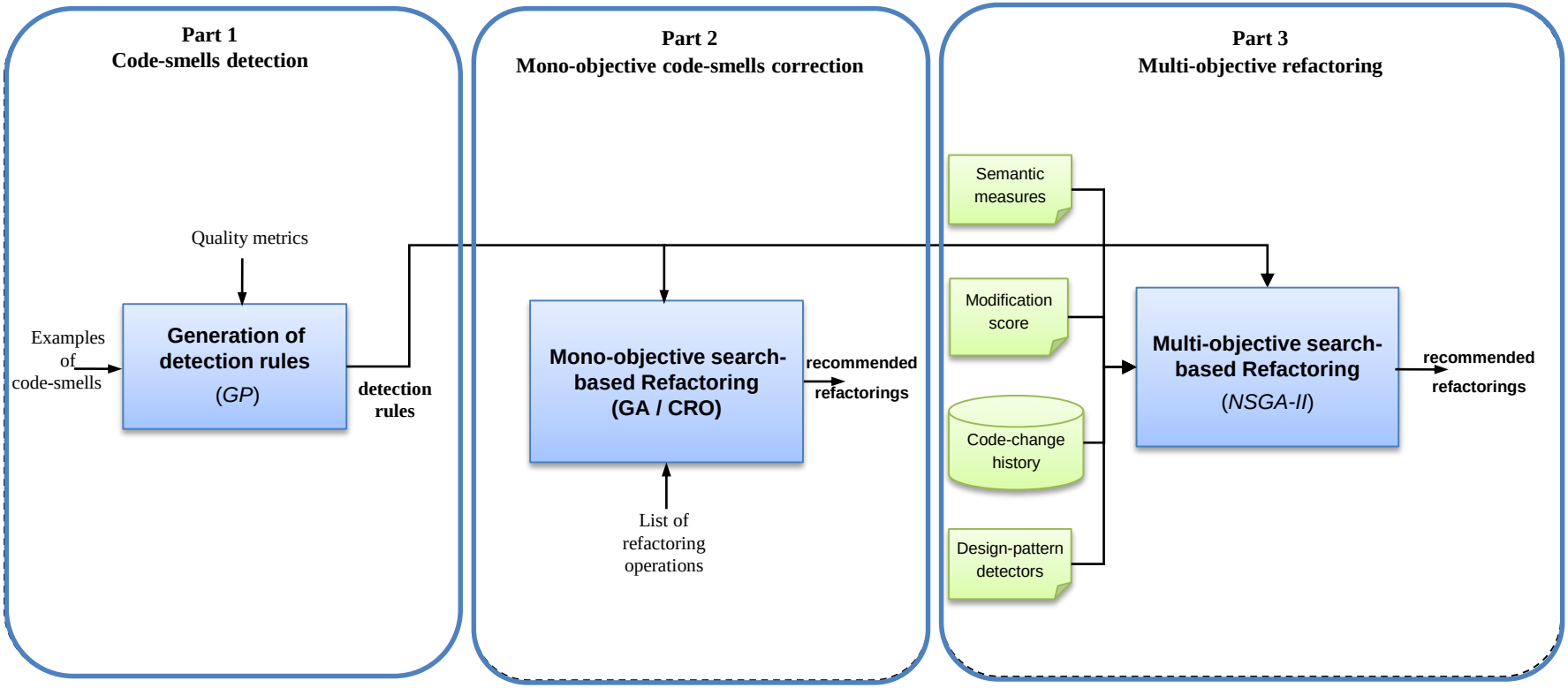
Thesis

- Goal

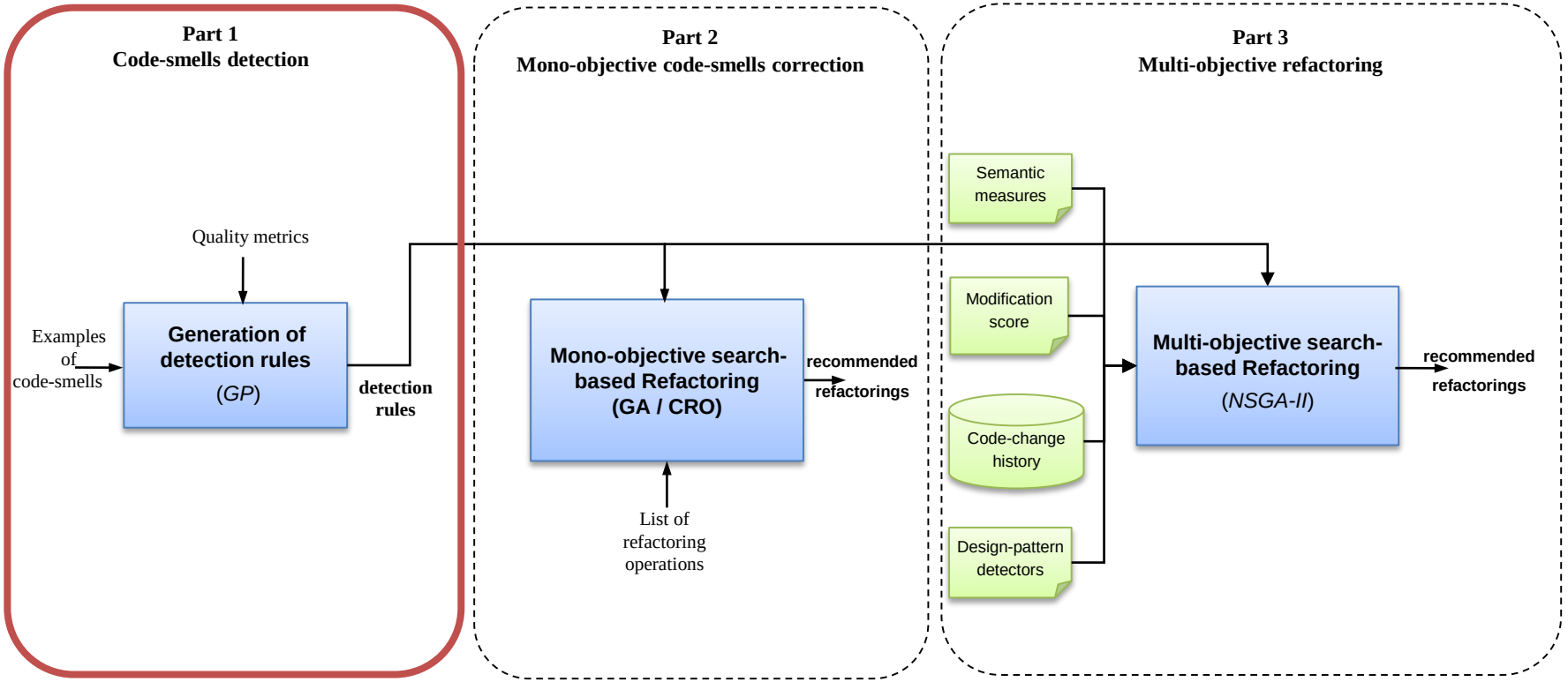
"Automated approach for recommending software refactoring"

- Generate code-smells detection rules
- Find refactoring solutions to fix code-smells while preserving the design semantics

Research methodology



Research methodology



Code-smells detection

- Existing work
  - Manual (Brown et al. ‘98, Fowler and Beck ‘99)
  - Metrics-based (Palomba et al ‘13, Marinescu et al. ’04, Salehie et al. ’06, Maiga et al.‘12)
  - Visual (Dhambri et al. ’08, Langelier et al. ’05)
  - Symptoms-based (Moha et al. ’08, Murno et al. ‘08)

Definition ➔ symptoms ➔ detection algorithm

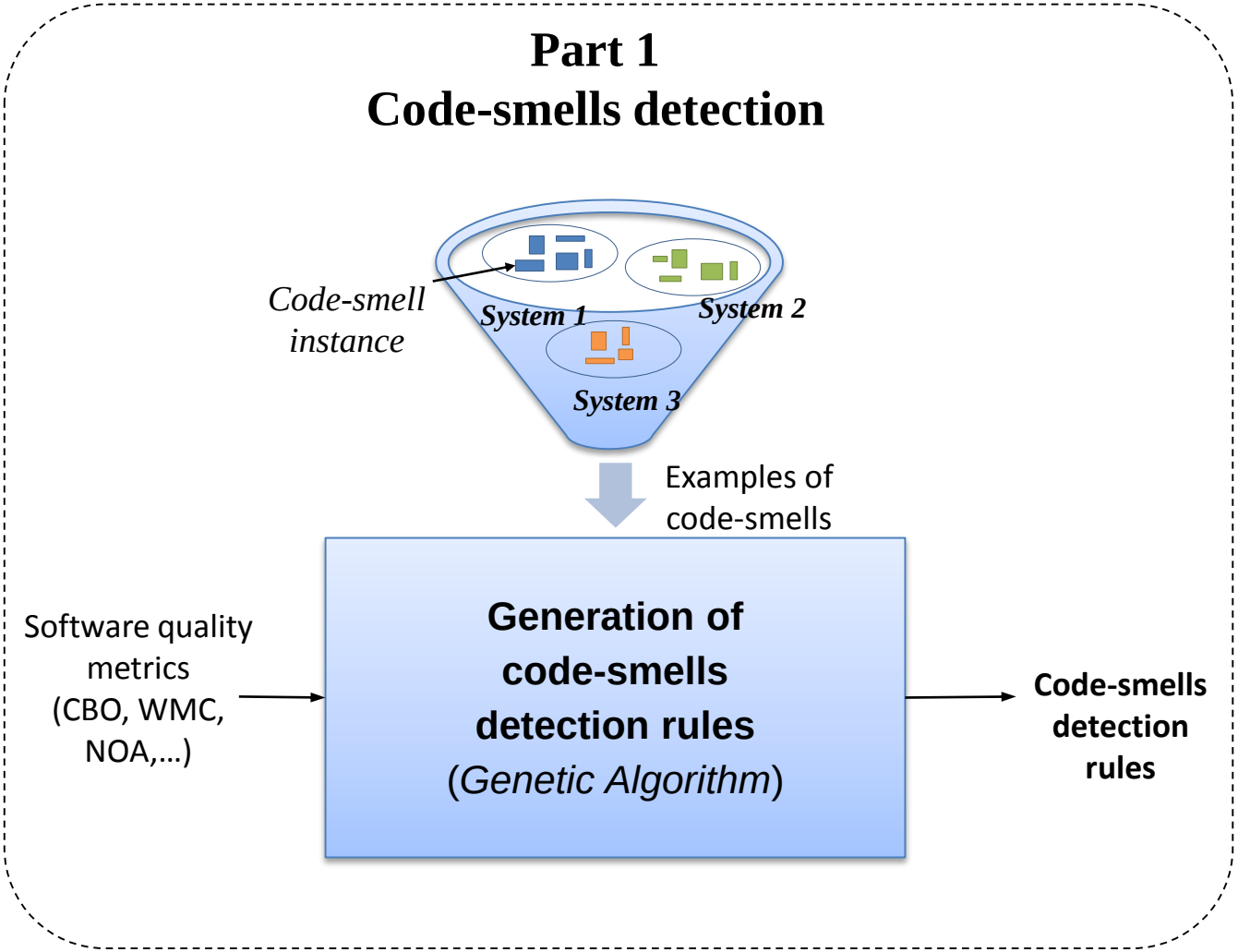
Problem statement

- Difficult to define/express detection rules
  - Large list of code-smells to categorize
  - Large exhaustive list of quality metrics
  - Huge number of possible threshold values
  - Huge space to explore: An expert to manually write and validate detection rules

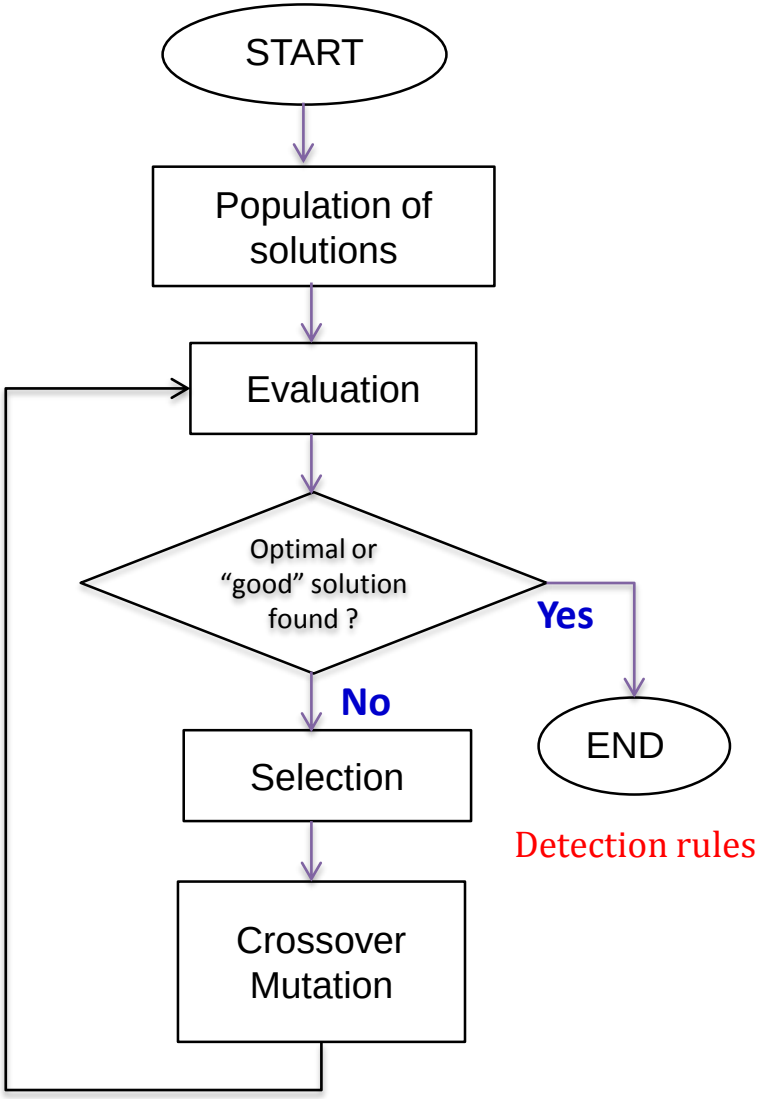
- Difficult to derive consensual rules
  - Diverge expert's opinions
  - No consensual definition of symptoms
  - The same symptom could be associated to many code-smell types
  - Easier to describe examples than translating symptoms into rules

Idea: Infer detection rules from code-smell examples

Approach overview



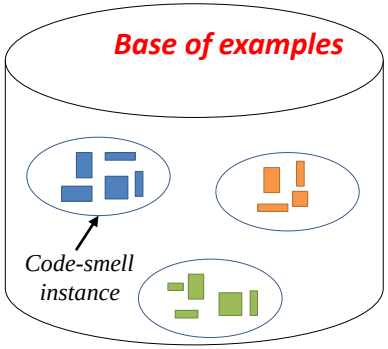
Genetic Programming



- Key elements
  - Representing of individual
  - Evaluating an individual
  - Deriving new individuals using genetic operators

GP adaptation

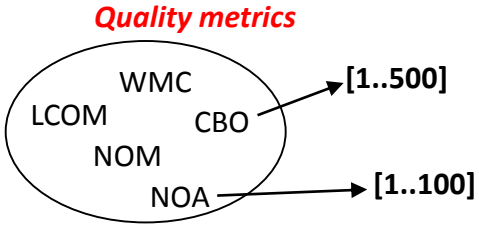
Solution representation



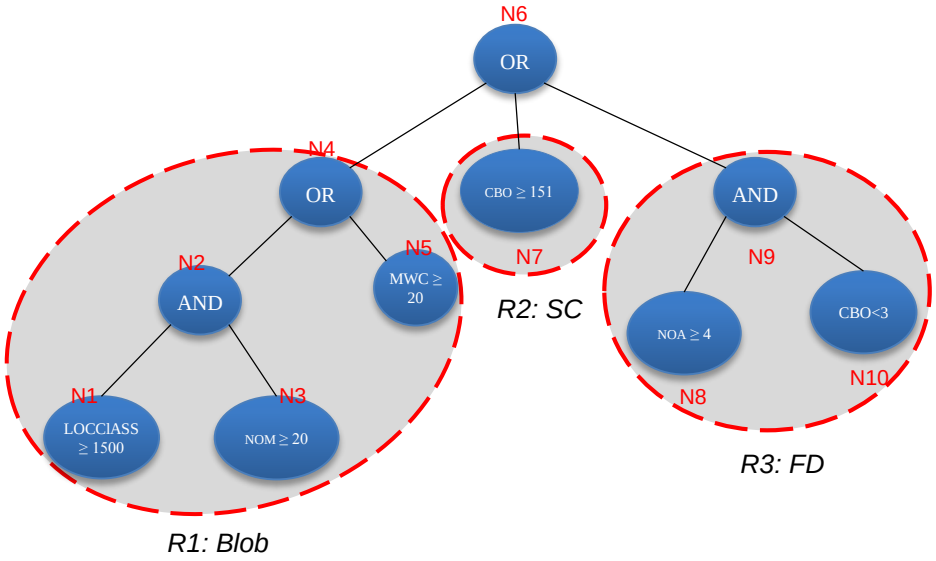
1 : Blob → `1 : If (LOCCLASS≥1500) AND (NOM≥20) OR (WMC>20) Then Blob`

2 : Spaghetti code → `2 : If (CBO≥ 151) Then Spaghetti code`

3 : Functional decomposition → `3 : If (NOA≥4) AND (WMC<3) Then Functional decomposition`



Tree representation



Fitness function

$$Recall = \frac{true\ positives}{total\ number\ of\ code\ smells}$$

$$Precision = \frac{true\ positives}{number\ of\ detected\ code\ smells}$$

Evaluation: detection results

- Studied systems

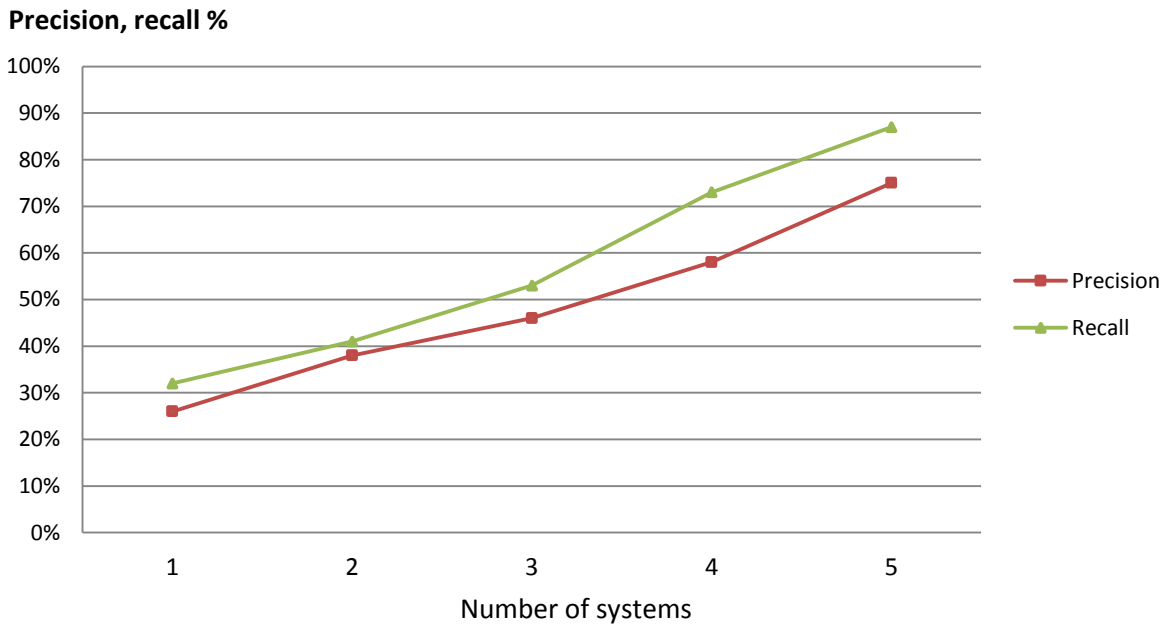
Systems	# of classes	KLOC	# of code-smells
Quick UML v2001	142	19	11
LOG4J v1.2.1	189	21	17
GanttProject v1.10.2	245	31	41
Xerces-J v2.7.0	991	240	66
ArgoUML v0.19.8	1230	1160	89
AZUREUS v2.3.0.6	1449	42	93

- Three types of code-smells
  - Blob, Spaghetti code, Functional decomposition
- 6-fold cross validation
  - Detect smells in a system using the 5 other systems
- Detection precision and recall

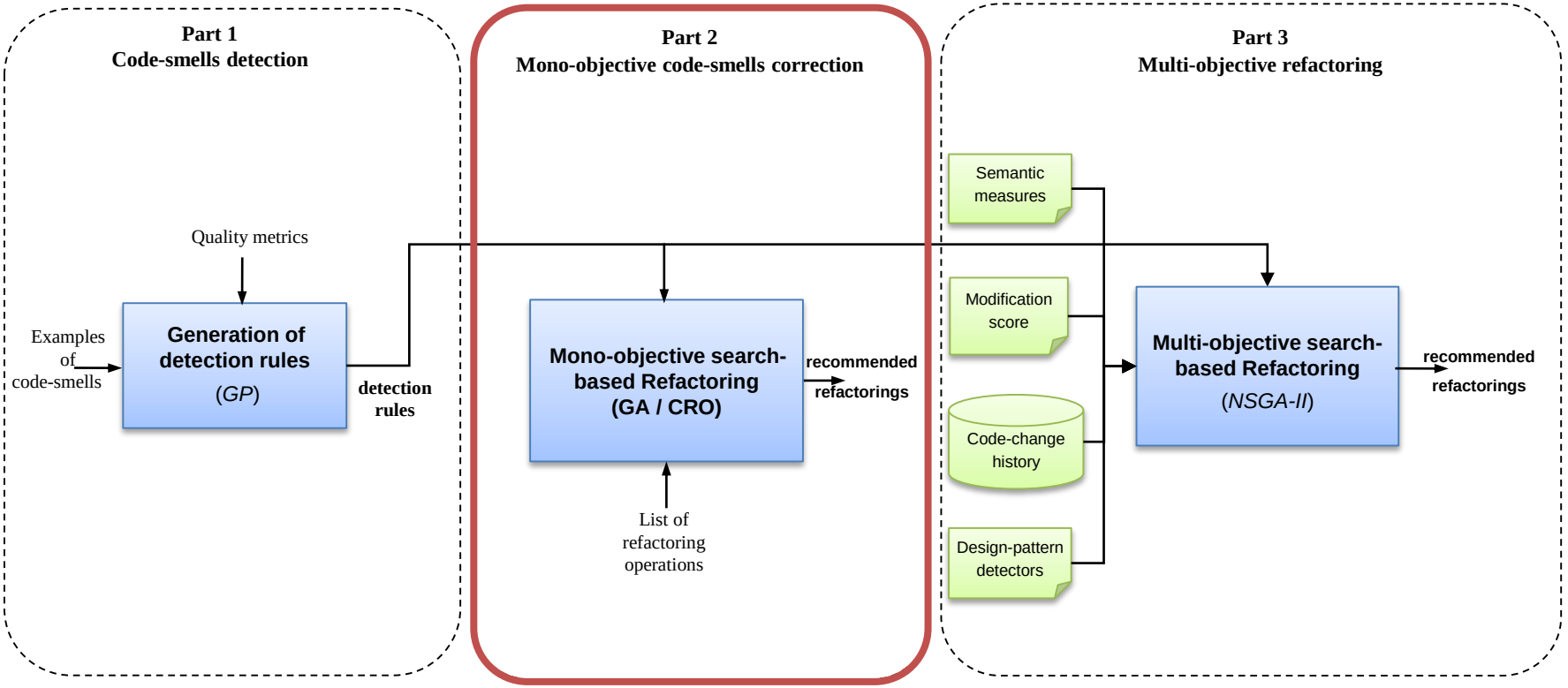
Detection results

System	Precision	Recall
GanttProject	Blob : 100% SC : 93% FD : 91%	100% 97% 94%
Xerces-J	Blob : 97% SC: 90% FD: 88%	100% 88% 86%
ArgoUML	Blob : 93% SC: 88% FD: 82%	100% 91% 89%
QuickUML	Blob : 94% SC: 84% FD: 81%	98% 93% 88%
AZUREUS	Blob : 82% SC: 71% FD: 68%	94% 81% 86%
LOG4J	Blob : 87% SC: 84% FD: 66%	90% 84% 74%
Average	*Blob : 92%* *SC: 85%* *FD: 79%*	*97%* *89%* *86%*

Examples-size variation



Outline


Part 1
Code-smells detection
Quality metrics
Examples of code-smells
Generation of detection rules (GP)
detection rules
Part 2
Mono-objective code-smells correction
Mono-objective search-based Refactoring (GA / CRO)
recommended refactorings
List of refactoring operations
Part 3
Multi-objective refactoring
Semantic measures
Modification score
Code-change history
Design-pattern detectors
Multi-objective search-based Refactoring (NSGA-II)
recommended refactorings

Existing work on refactoring

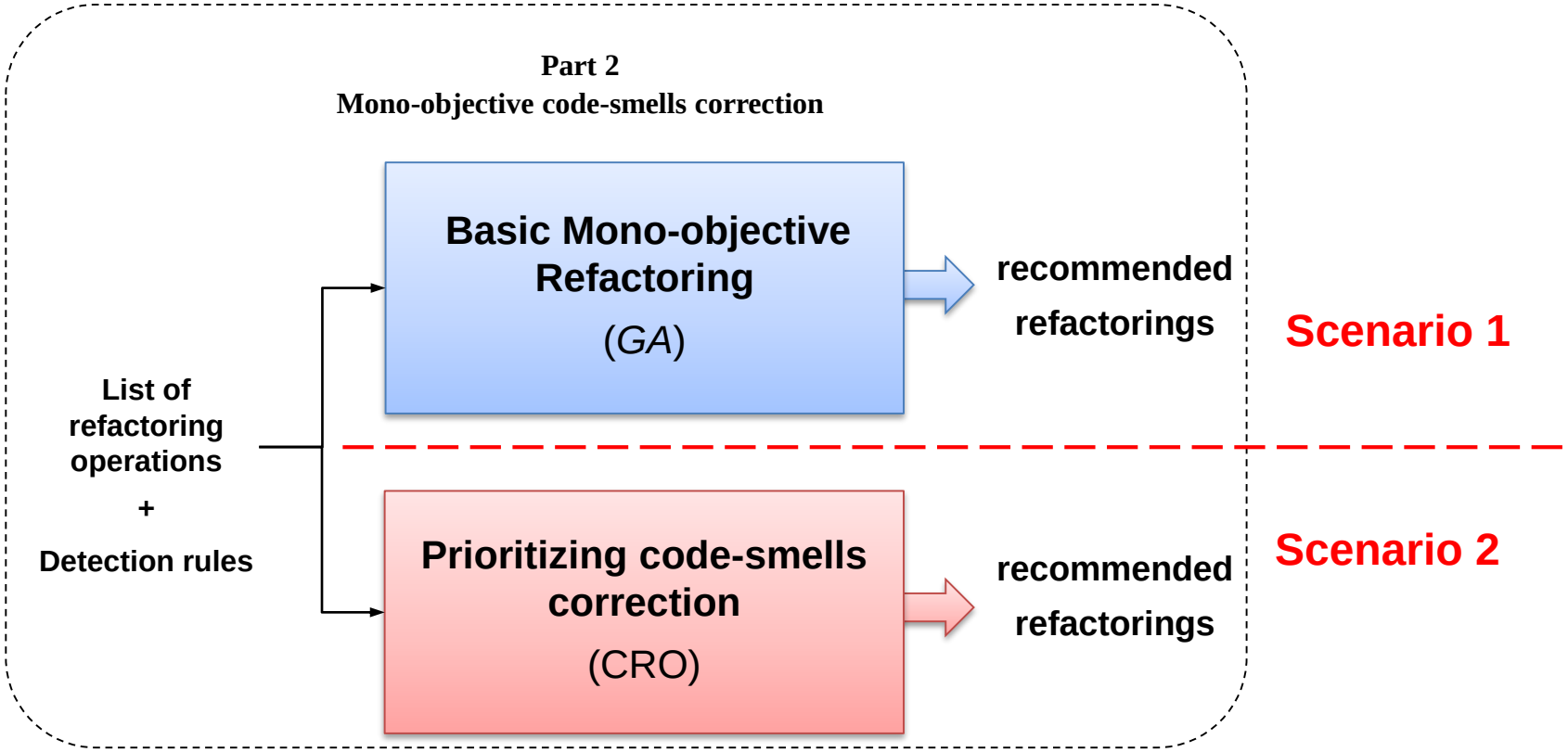
- Manual approaches
  - Standard refactorings (Fowler et al. '98)
- Metric-based approaches
  - Search-based techniques
    - Find the best sequence of refactorings (Seng et al. '06, Harman et al. '07, O'Keeffe et al. '08,)
  - Analytic approaches
    - Study of relations between some quality metrics and refactoring changes (Sahraoui et al. '00, Du Bois et al. '04, Moha et al. '08)
- Clustering-based approaches
  - Fix code-smells (Marios et al, '11, JDeodorant)

Problem statement

- Difficult to define "standard" refactorings
  - Pre-define refactoring solutions for each code-smell type
- Difficult to establish the link between refactoring and quality improvement
- Improving some quality metrics do not fix code-smells
- Is not practical to correct code-smells separately
  - Correcting a code-smell may produce other code-smells
  - Do not consider the impact of refactoring
- Do not take into consideration the semantic coherence dimension

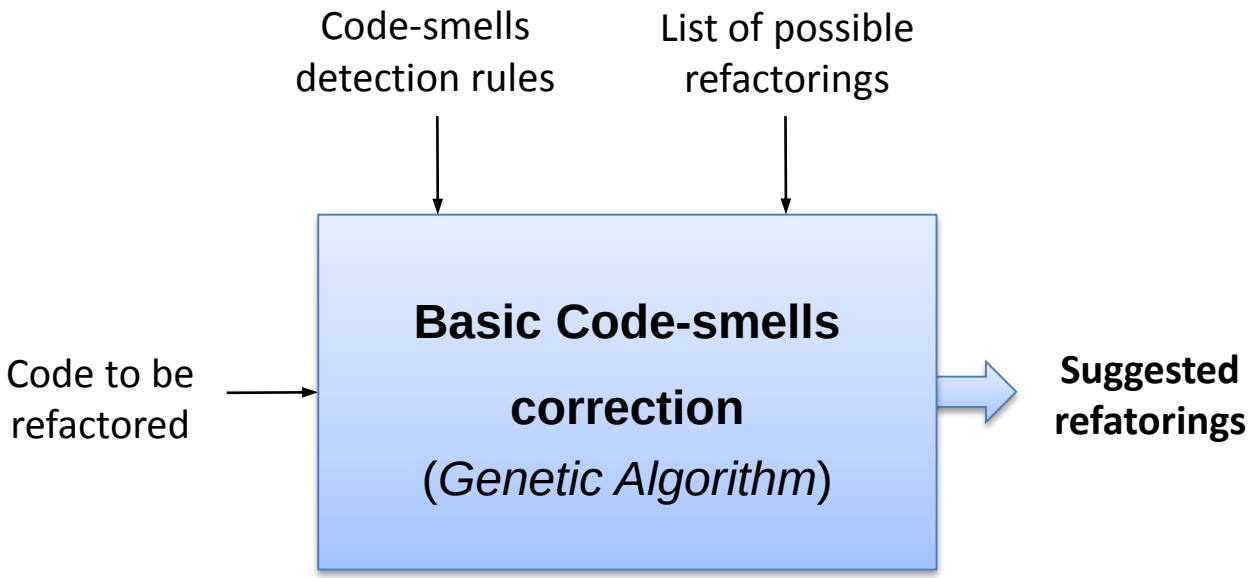
Mono-objective Refactoring

- Two scenarios
  1. Enough time/resources
  2. Time/resources limitations



Basic mono-objective refactoring

Scenario 1



Genetic algorithm adaptation

- Genetic Algorithm
  - Solution representation
    - Individual = Sequence of refactoring operations

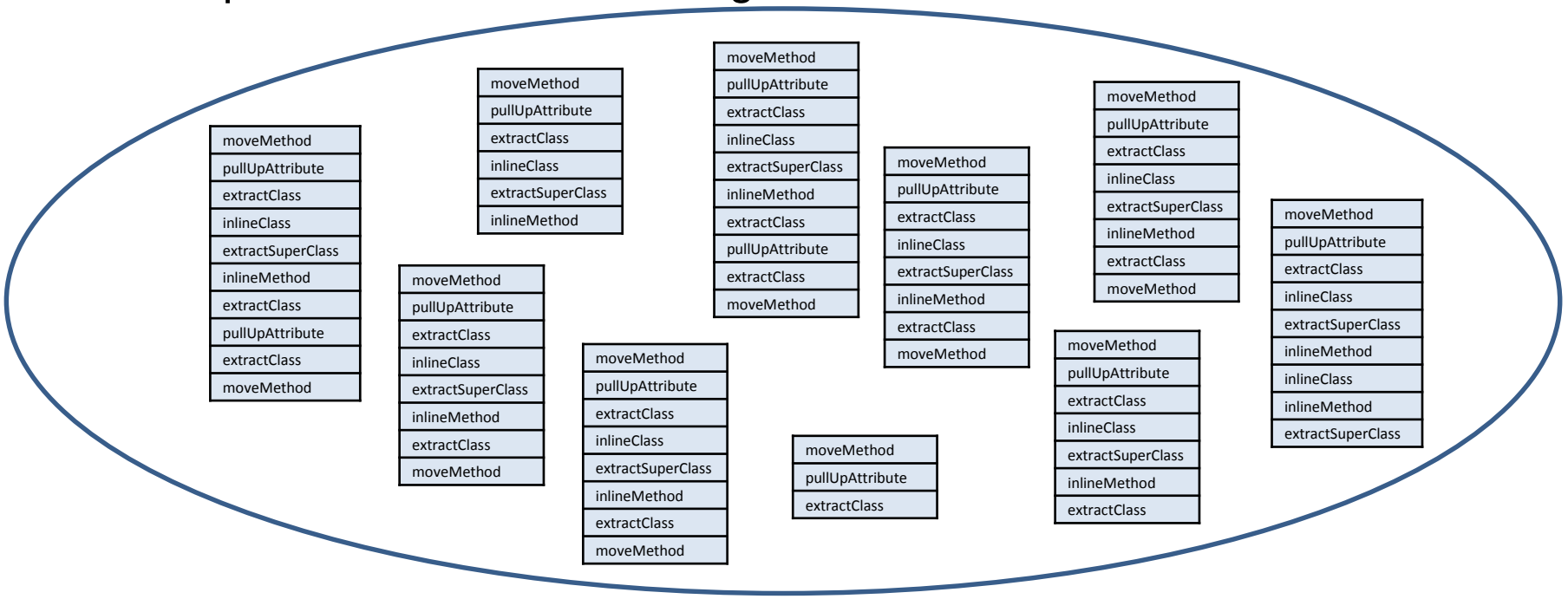
1	moveMethod
2	pullUpAttribute
3	extractClass
4	inlineClass
5	extractSuperClass
6	inlineMethod

    - 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)

Genetic algorithm adaptation

- Population creation
  - Population: set of refactoring solutions



- Fitness function

$$\text{Correction ratio} = \frac{\#\text{ code smells } after\ refactoring}{\#\text{ detected code smells}}$$

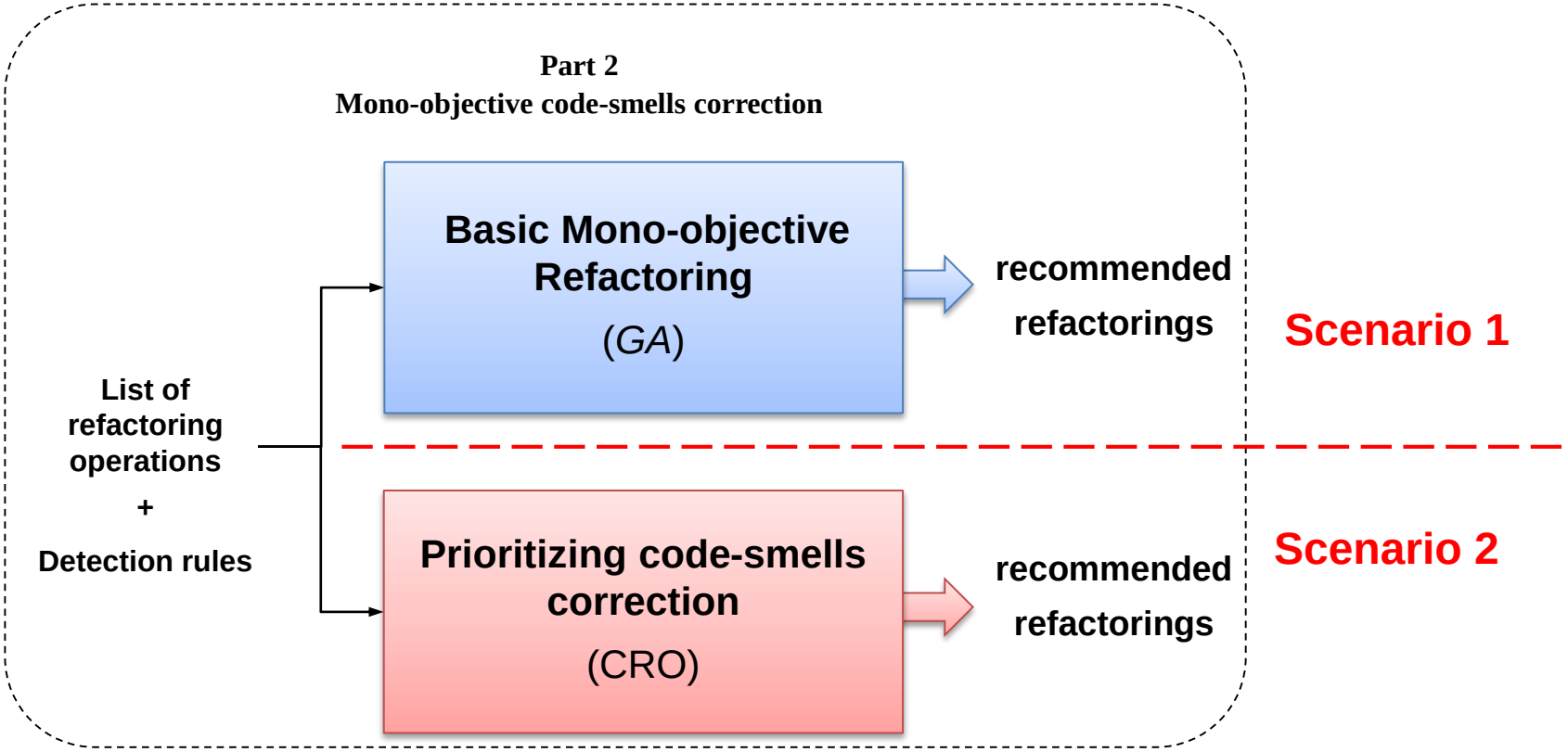
Evaluation: basic refactoring results

System	CCR	RP
GanttProject	95% (39|41)	52 %
Xerces-J	89% (59|66)	49 %
ArgoUML	85% (76|89)	59 %
QuickUML	90% (26|29)	59 %
LOG4J	88% (15|17)	51 %
AZUREUS	94% (87|93)	57 %
Average	*90%*	*54%*

$$\textbf{CCR} = \frac{\textbf{\# code smells after applying refactorings}}{\textbf{\# code smells before refactoring}}$$

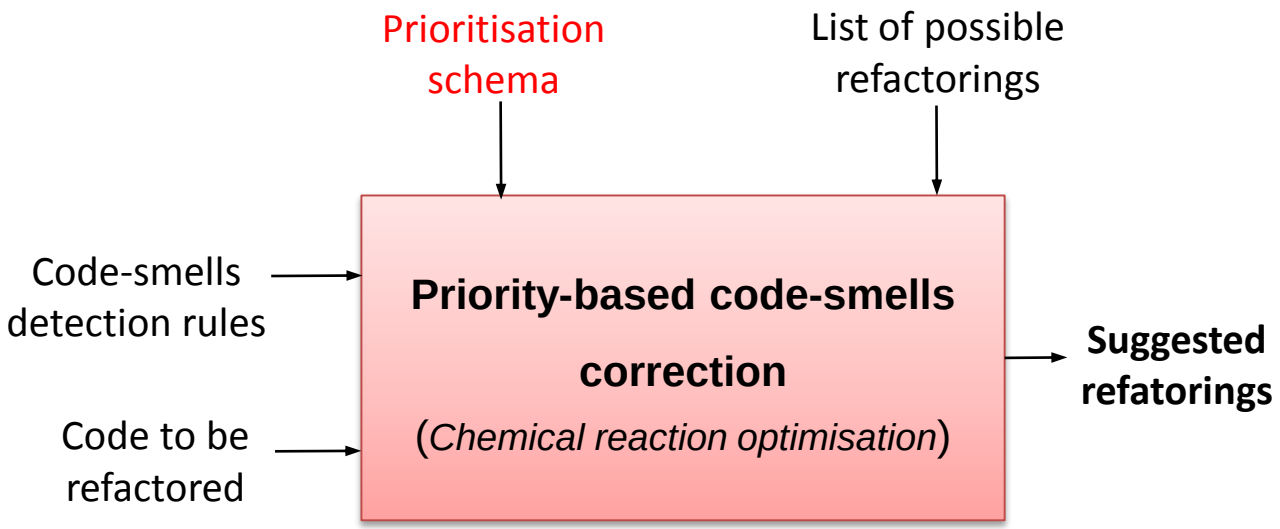
$$\textbf{RP} = \frac{\textbf{\# meaningful refactorings}}{\textbf{\# proposed refactorings}}$$

Mono-objective Refactoring



Priority-based refactoring

- Second scenario
  - Time / resources limitations
  - Don’t need to fix all code-smells -> only the most critical ones
  - Idea: Prioritize the correction of code smells



Prioritizing code-smells correction

- Four heuristics
  - Priority
    - rank code-smells according to the maintainers preferences
  - Severity
    - rank code-smells according to a set of design-properties (size, complexity, coupling, cohesion, hierarchy, etc.)
  - Risk
    - the risk score corresponds to the deviation from good design practices.
    - the more code deviates from good practices, the more it is likely to be risky
  - Importance
    - the importance of a class corresponds to their change frequency
    - the more a class undergoes changes, the more it is likely to be problematic
    - if a code-smell remains unmodified, the system may not experience problems

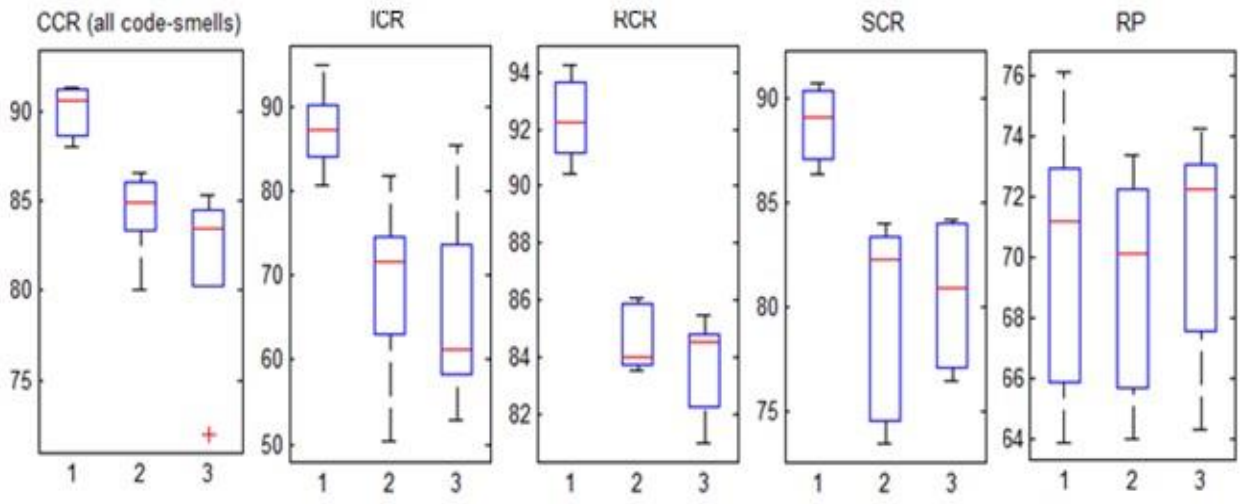
Prioritizing code-smells correction

- Goal:
  - maximize the correction of the critical code-smells
- Objective function
  - weighted sum of the prioritization heuristics

- Chemical reaction optimisation (CRO)
  - Inspired by the phenomenon of chemical reactions
  - Molecule <--> solution
  - Potential energy <--> objective function value
  - Collision <--> change operator (mechanism to find new solution)

Validation: CRO refactoring results

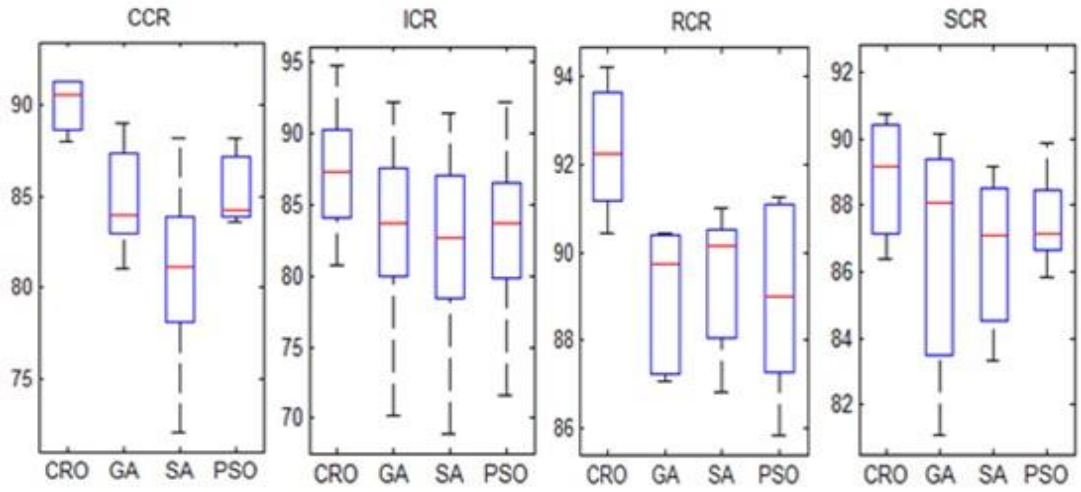
- Comparison
  1. Our CRO-based approach
  2. CRO without prioritization
  3. Basic GA approach



Refactoring comparison results for the five systems from 31 simulation runs

SBSE validation

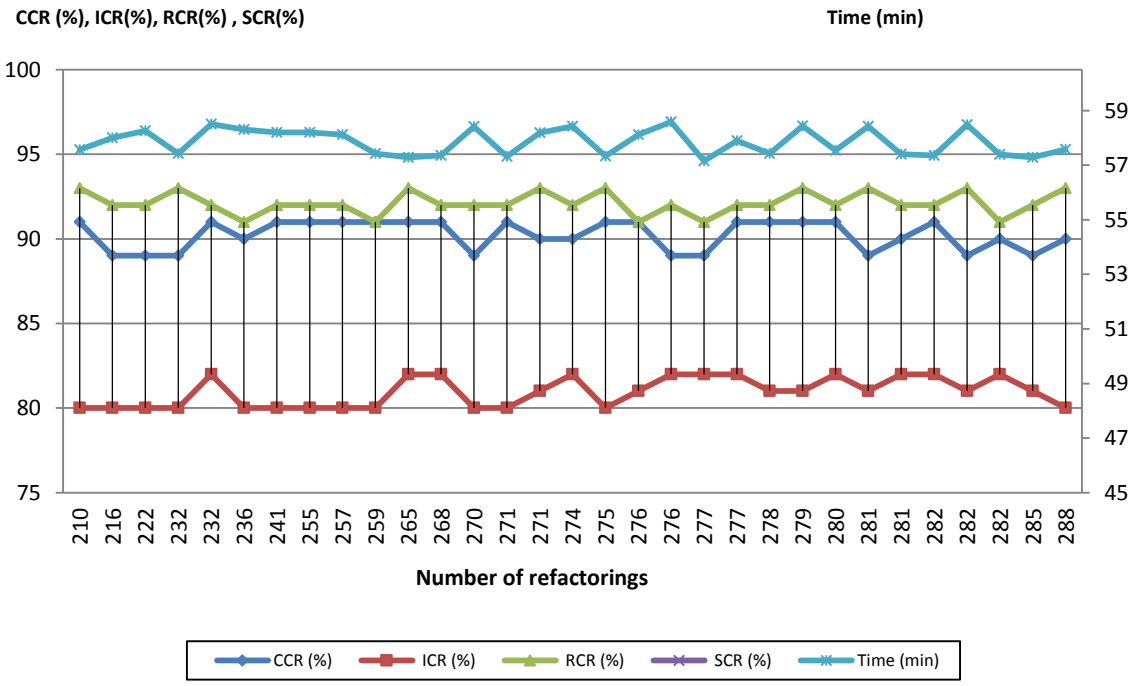
- Comparison
  1. Chemical Reaction Optimization (CRO)
  2. Genetic Algorithm (GA)
  3. Simulated Annealing (SA)
  4. Particle Swarm Optimisation (PSO)



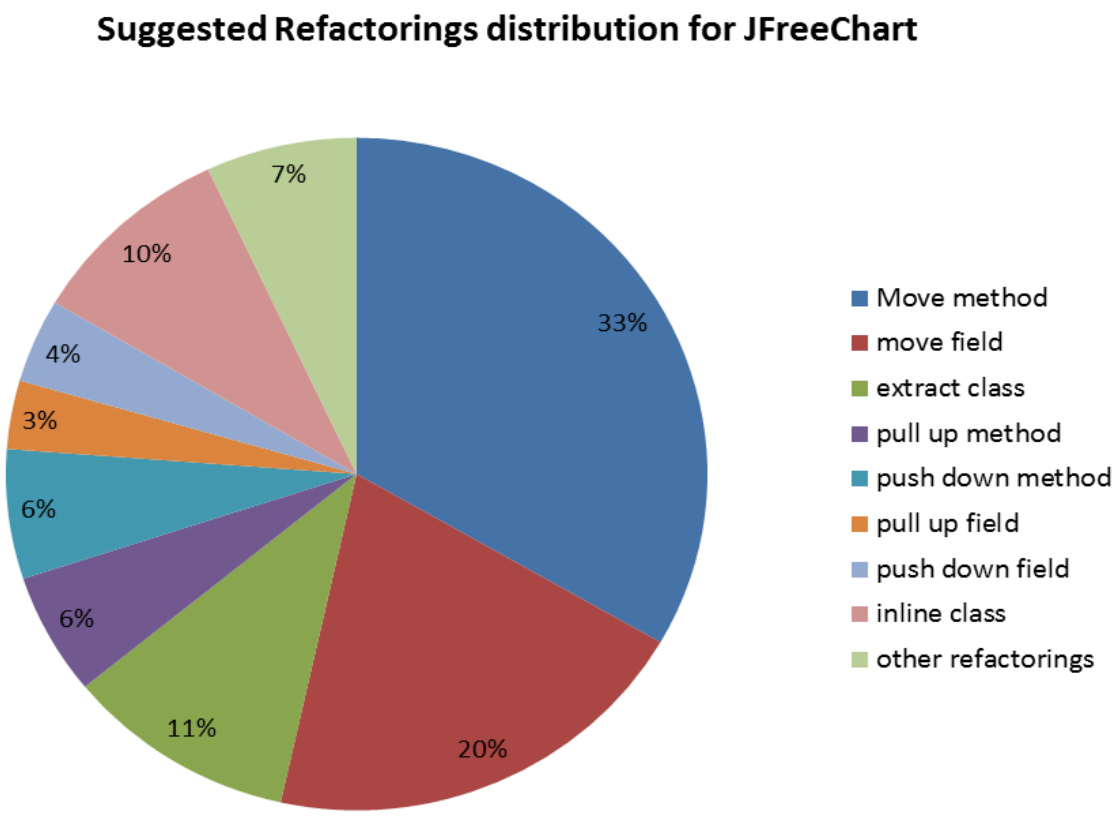
Refactoring comparison results for the five systems from 31 simulation runs

Results

- Stability



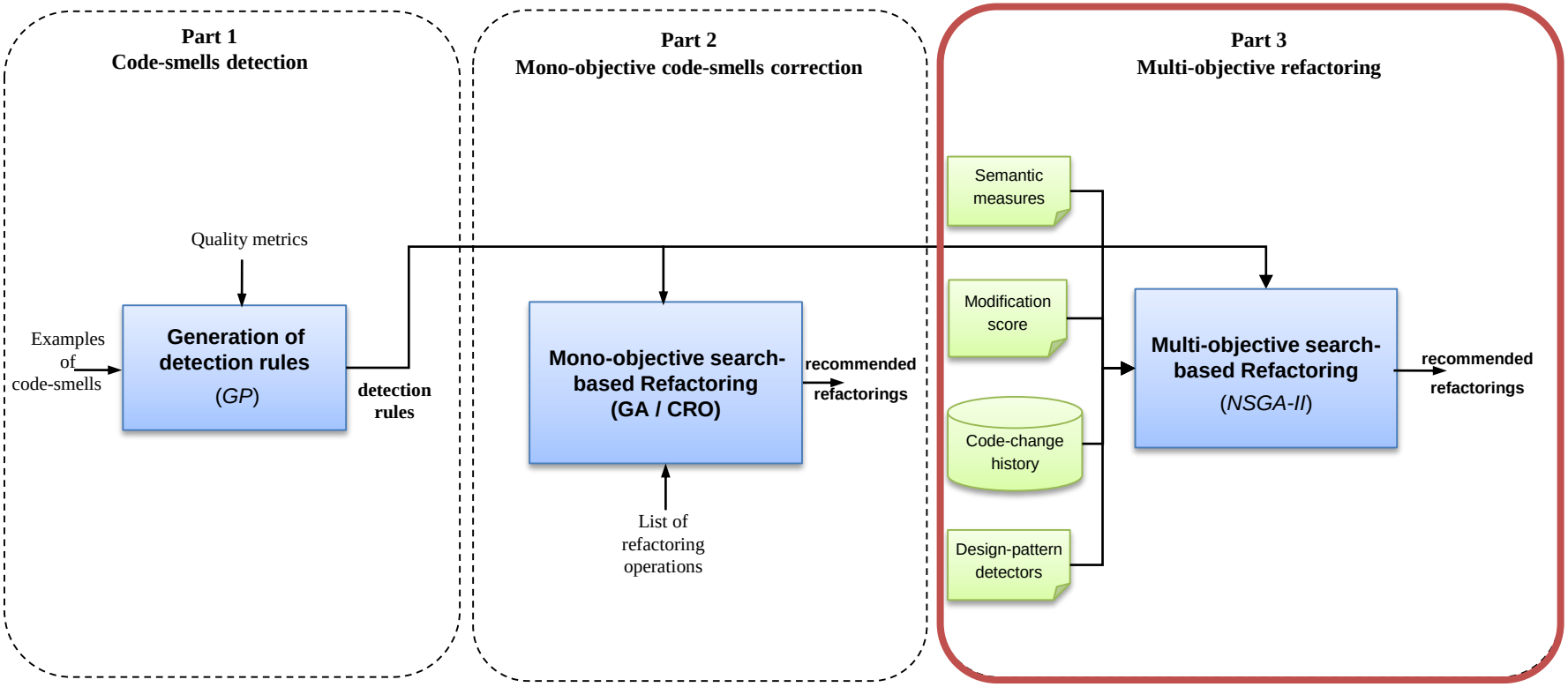
Refactorings distribution



Mono-objective formulation

- Advantages
  - Most of code-smells are fixed
  - Critical code-smells are fixed first (prioritized)
- Limitations
  - Not all suggested refactoring operations are semantically feasible
  - Some refactoring solutions requires a considerable number of changes
- Single perspective ?

Outline


Part 1
Code-smells detection
Part 2
Mono-objective code-smells correction
Part 3
Multi-objective refactoring
Quality metrics
Examples of code-smells
Generation of detection rules (GP)
detection rules
Mono-objective search-based Refactoring (GA / CRO)
recommended refactorings
List of refactoring operations
Semantic measures
Modification score
Code-change history
Design-pattern detectors
Multi-objective search-based Refactoring (NSGA-II)
recommended refactorings

Multi-objective formulation

- Two intuitions
  - Design preservation objectives
    - Semantic approximation
    - Number of changes
    - Conformance with refactoring history

Application 1

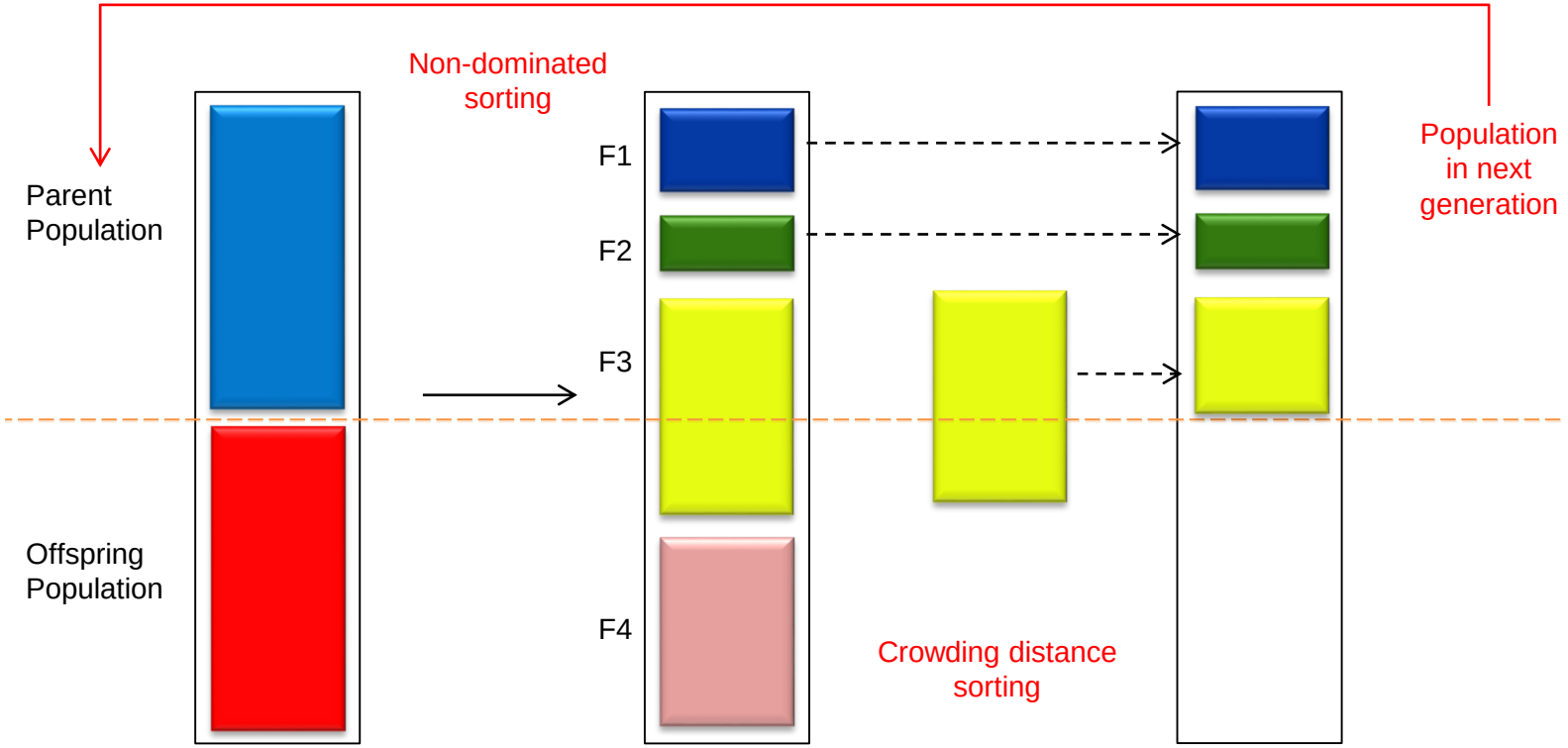
  - Quality objectives
    - Fix code-smells
    - Improve quality indicators
    - Introduce design patterns
    - …

Application 2

Multi-objective optimization problem

NSGA-II overview

- NSGA-II: Non-dominated Sorting Genetic Algorithm (K. Deb et al., '02)



Multi-objective formulation

- Two intuitions

  - Design preservation objectives
    - Semantic approximation
    - Number of changes
    - Conformance with refactoring history

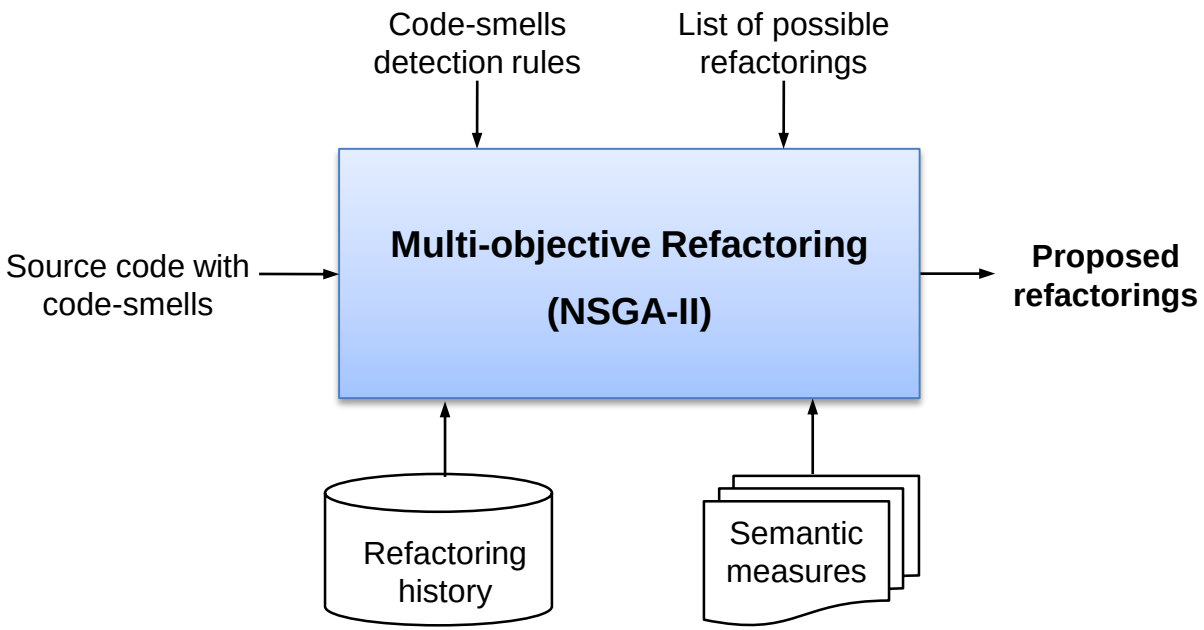
  Application 1

  - Quality objectives
    - Fix code-smells
    - Improve quality indicators
    - Introduce design patterns
    - …

  Application 2

Multi-objective optimization problem

Application 1: Design preservation FIRST !

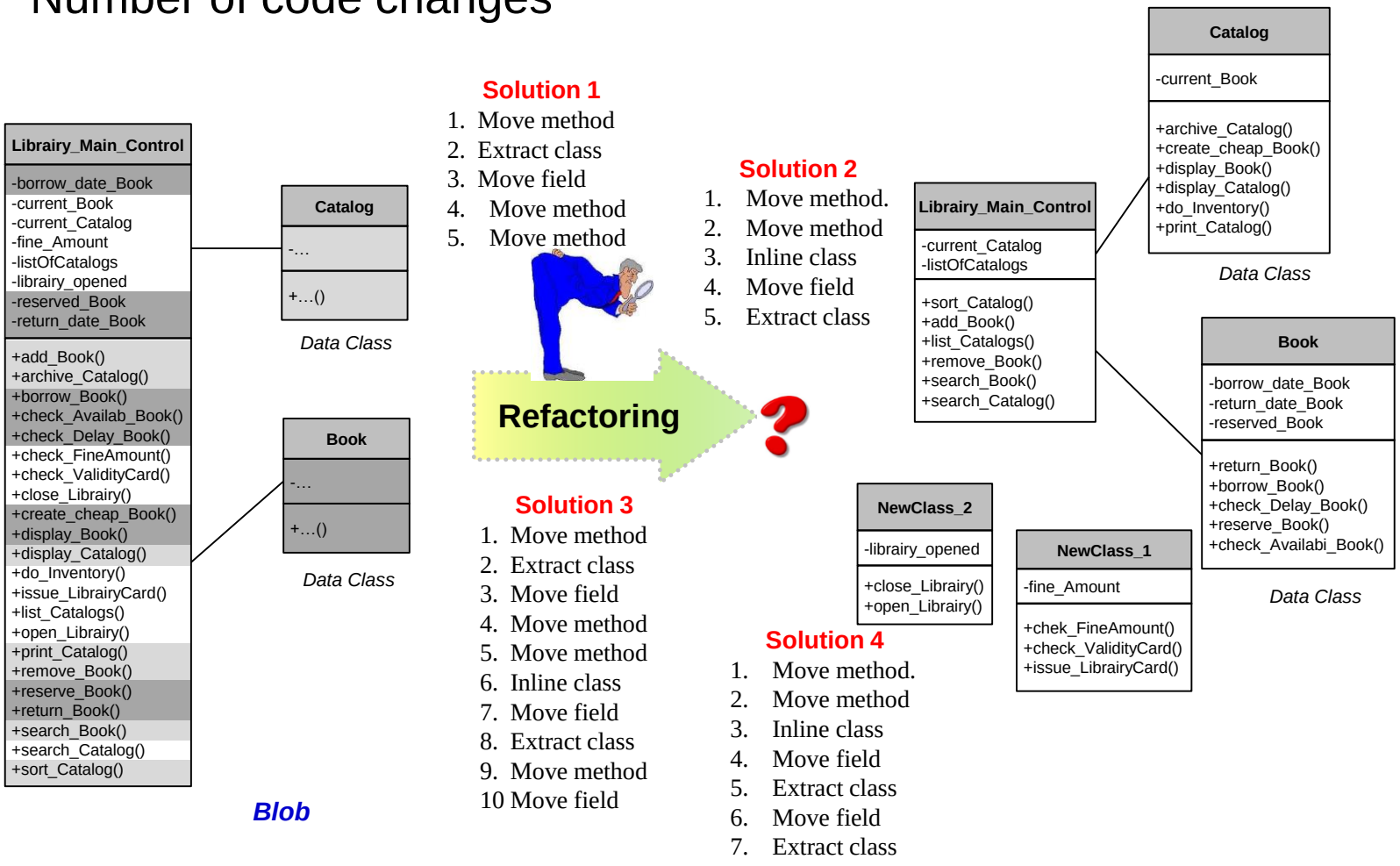


NSGA-II adaptation

- Four objective functions
  1. Quality
     - Calculate the number of fixed code-smells
  2. Code changes
     - Calculate the number of atomic changes required when applying refactoring
  3. Approximate semantic similarity
     - Calculate cosine similarity between vocabulary used
  4. Maintain the conformance with change history
     - Calculate a similarity score between the a suggested refactoring and a base of refactorings applied in the past

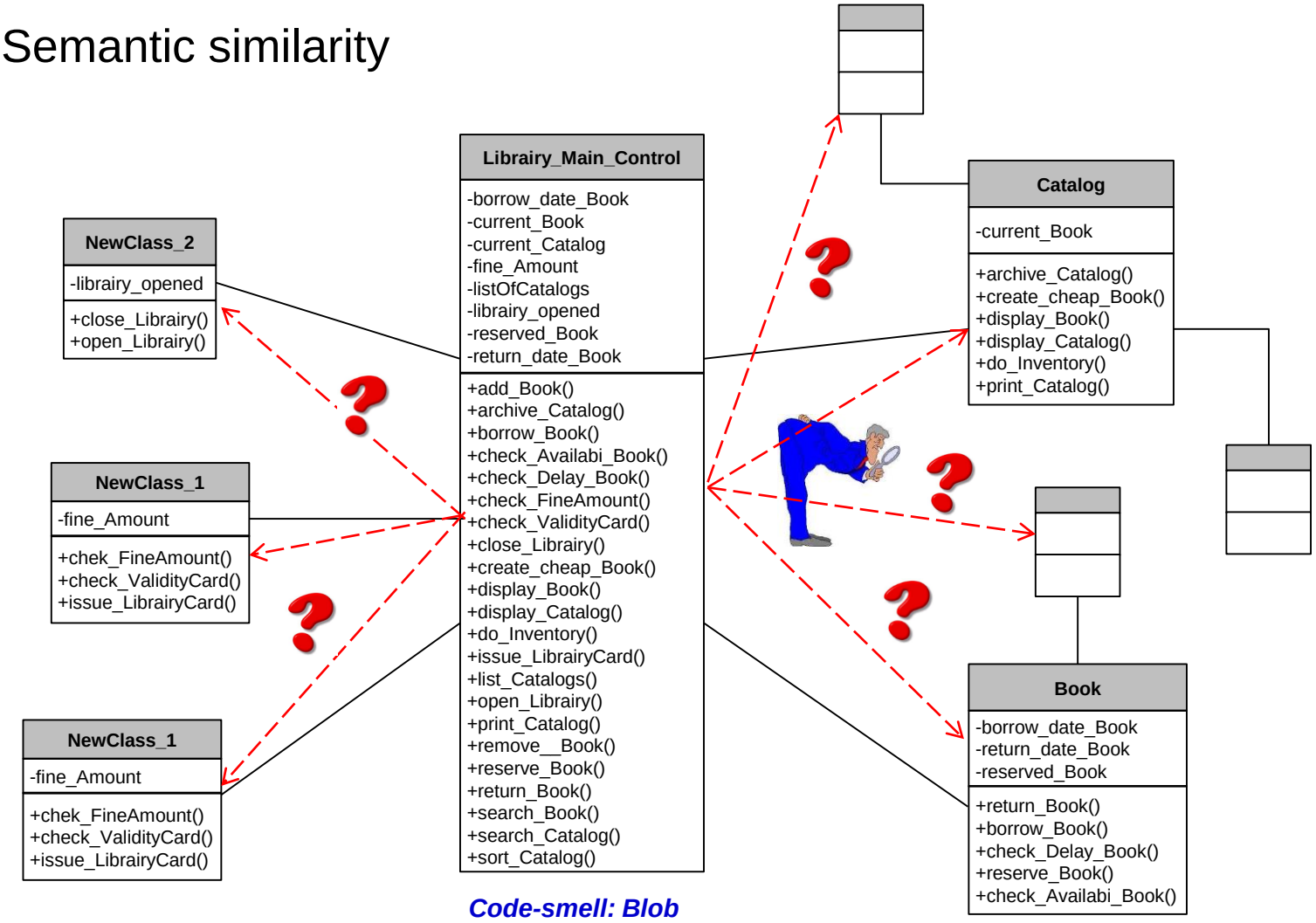
Refactoring criteria. . .

- Number of code changes



Refactoring criteria. . .

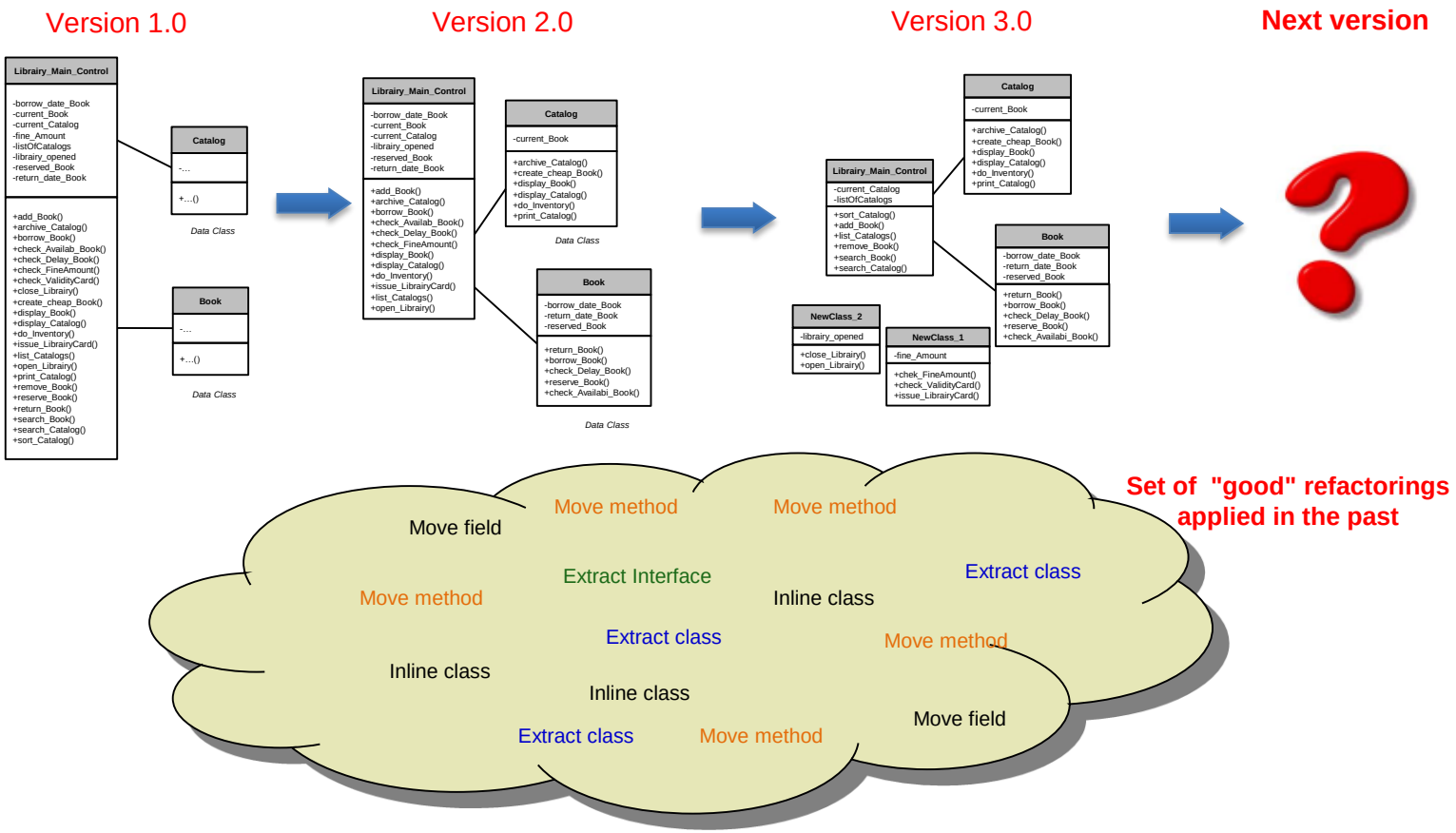
- Semantic similarity



Code-smell: Blob

Refactoring criteria. . .

- Conformance with refactorings applied in the past



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, and Functional decomposition
- Data
  - Collect refactorings from previous versions (Ref-Finder)
  - Manual inspection

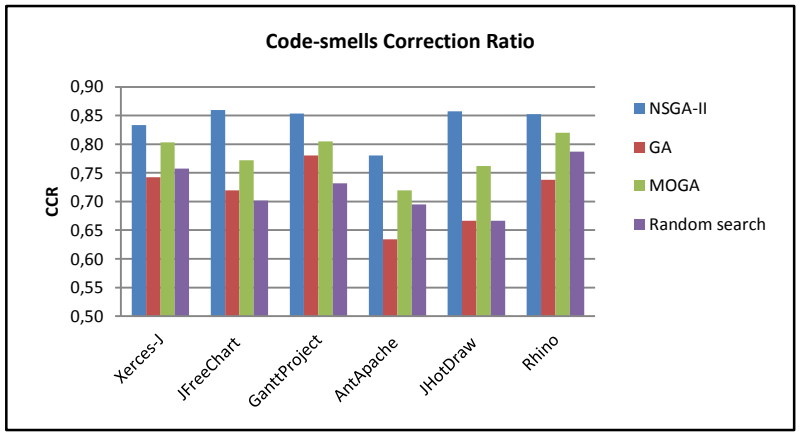
Empirical evaluation

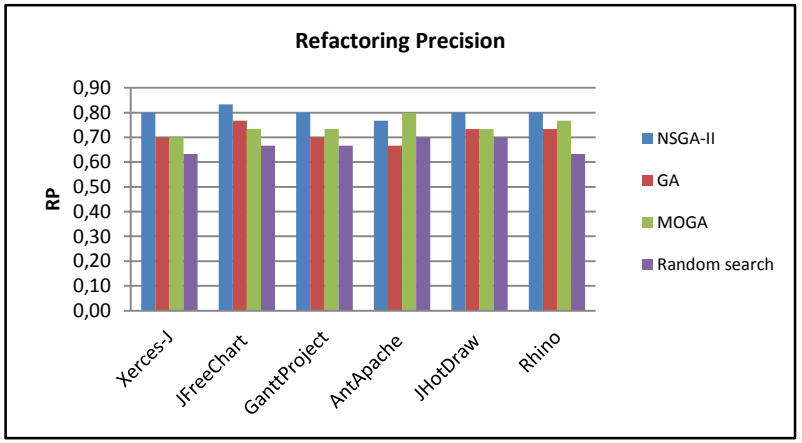
- Survey
  - Questionnaire: evaluate the suggested refactorings
  - Sample of 10 refactoring operations
  - 18 subjects
    - graduate/undergraduate students, assistant professors, junior software developers
    - Subjects are volunteers and familiar with java programming
    - 2 to 13 years experience on Java programming
    - 6 groups
- Comparison to state-of-the-art research
  - Harman et al 2007, Basic GA approach
- Comparison to other mono and multi-objective algorithms
  - MOGA, GA, Random Search

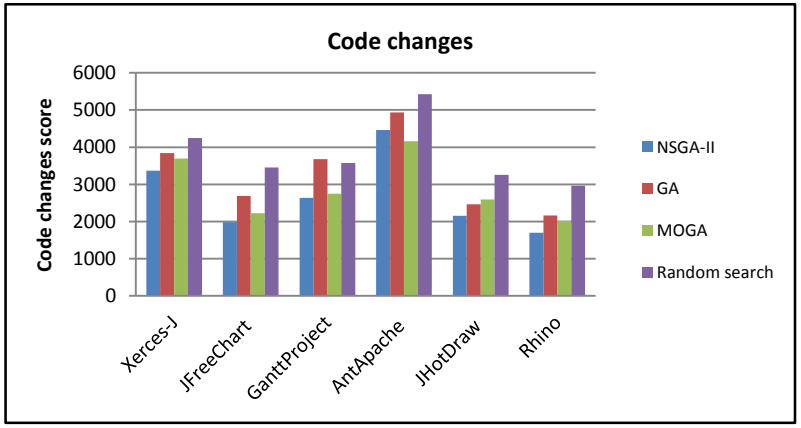
Multi-objective refactoring results

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

SBSE validation







Multi-objective formulation

- Two intuitions
  - Design preservation objectives
    - Semantic approximation
    - Number of changes
    - Conformance with refactoring history

Application 1

  - Quality objectives
    - Fix code-smells
    - Improve quality indicators
    - Introduce design patterns
    - …

Application 2

Multi-objective optimization problem

Application 2: Quality Improvement

- Quality FIRST
- Objectives functions
  1. Fix code-smells
  2. Improve quality indicators
  3. Introduce design patterns
- Design preservation
  - Constraints to satisfy when applying refactorings

Evaluation: Application 2

- Studied systems

Systems	Release	# classes	KLOC	# code-smells	# design patterns
Xerces-J	v2.7.0	991	240	81	36
GanttProject	v1.10.2	245	41	49	15
AntApache	v1.8.2	1191	255	92	38
JHotDraw	v 6.1	585	21	24	18

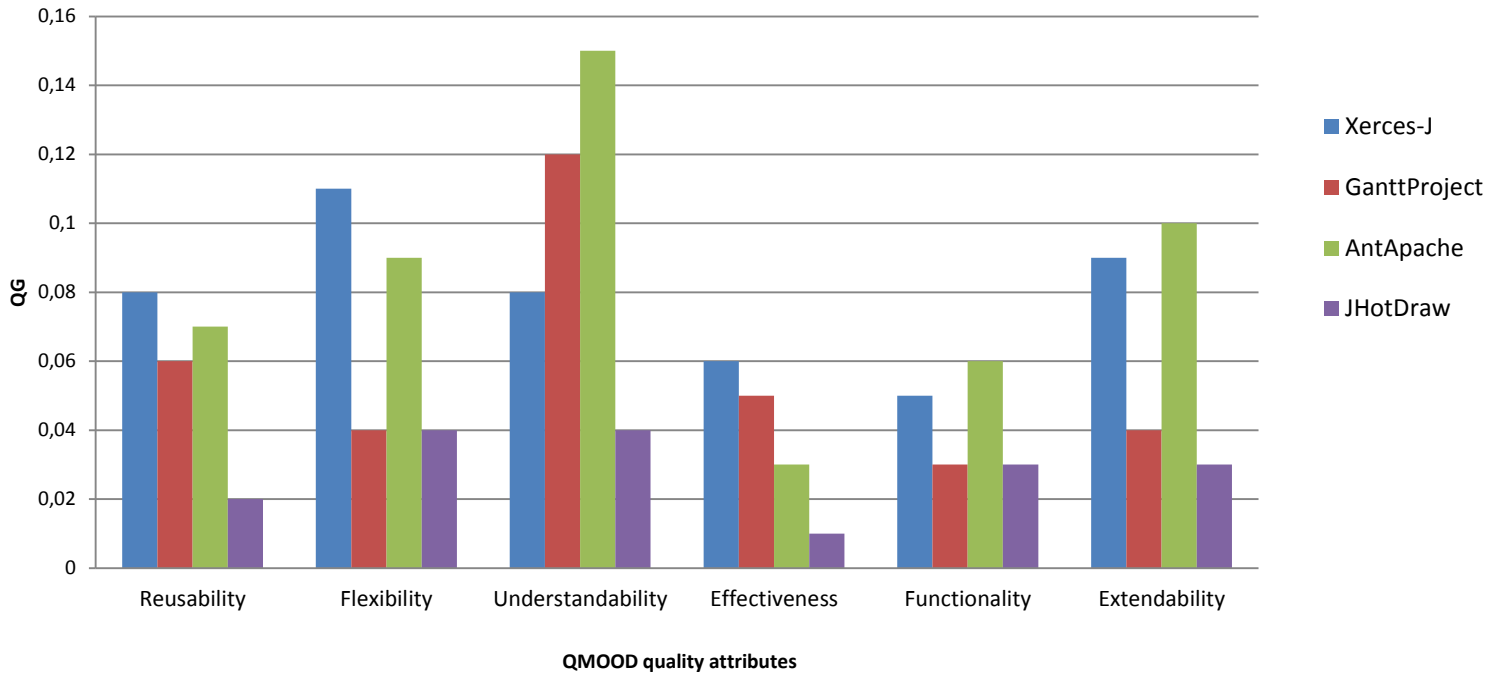
- Code-smells
  - God class, Feature Envy, Data Class, and Spaghetti Code
- Design patterns
  - Visitor, Factory method, singleton
- Comparison with existing approahes
  - Seng et al. ’06, Jensen et al. ’11, Basic GA approach

Refactoring results

Systems	Approaches	CCR	NP	QG
Xerces-J	Application 2	89%	12	0.47
	Seng et al.	23%	0	0.54
	Jensen et al.	14%	31	0.41
	Basic GA approach	88%	0	0.32
GanttProject	Application 2	88%	7	0.34
	Seng et al.	24%	1	0.33
	Jensen et al.	33%	14	0.35
	Basic GA approach	84%	0	0.21
AntApache	Application 2	86%	4	0.5
	Seng et al.	7%	0	0.52
	Jensen et al.	12%	28	0.51
	Basic GA approach	87%	0	0.39
JHotDraw	Application 2	83%	4	0.17
	Seng et al.	38%	0	0.19
	Jensen et al.	25%	9	0.14
	Basic GA approach	88%	0	0.1
Average (all systems)	Application 2	86%	7	0.37
	Seng et al.	23%	0.25	0.39
	Jensen et al.	21%	20.5	0.35
	Basic GA approach	86%	0	0.25

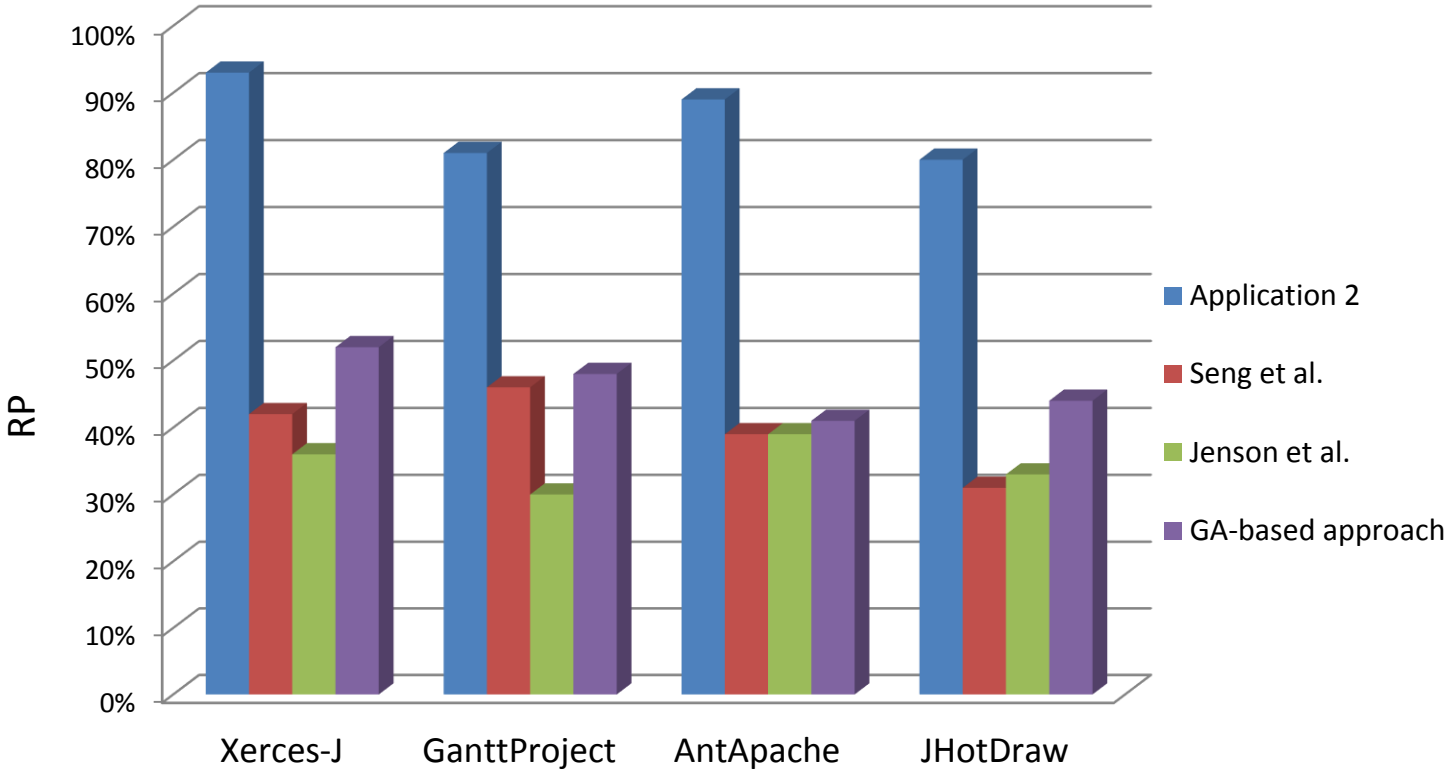
Results

- Quality improvement



Results

- Refactoring meaningfulness



Outline

- Context and problem
- Research methodology
- Code-smells detection
- Mono-objective software refactoring
- Multi-objective software refactoring
- Conclusion and perspectives

Conclusion

- First search-based code-smells detection approach
  - Search-based approach : GP
  - Infer detection rules from code-smell examples
- First mono-objective search-based code-smells correction
  - GA: maximize the number of fixed code-smells
  - CRO: Prioritize the correction of code-smells
- First multi-objective search-based refactoring approach
  - Application 1: preserve design
  - Application 2: improve quality
- Validation
  - Empirical evaluation
  - Very encouraging results
  - Comparison with existing approaches

Future research directions

- Short term
  - Code-smells detection
    - Consider the change history to improve the detection of code-smells
    - Consider other types of code-smells
    - Test our approach with other industrial systems
  - Refactoring
    - Provide a generic GUI-based tool to combine all the proposed approaches
    - Additional semantic constraints
    - Additional refactoring operations
  - An interactive component
    - Put the developer in the loop when recommending refactorings

Future research directions

- Long term
  - Detection of code-smells in SOA
  - Refactoring of Service-based systems
    - Define refactorings in the service level
    - Automatically recommend refactorings
  - Software migration
  - Empirical investigations
    - The correlation between code-smells and refactoring proneness

Publications

Journal Papers and Book Chapters

1. **Ali Ouni**, Marouane Kessentini, Slim Bechick and Houari Sahraoui, Prioritizing Code-smells Correction Tasks Using Chemical Reaction Optimization, Journal of Software Quality, 2014.

2. **Ali Ouni**, Marouane Kessentini, Mel Ó Cinnéide, Houari Sahraoui, Multi-criteria Software Refactoring: Quality, Code Changes, and Semantics Preservation, IEEE Transactions on Software Engineering, 2014. (under review).

3. Marouane Kessentini, **Ali Ouni**, Philip Langer, Manuel Wimmer and Slim Bechikh, Search-based Metamodel Matching with Structural and Syntactic Measures, Journal of Systems and Software (JSS), 2014.

4. Wael Kessentini, Marouane Kessentini, Houari Sahraoui, Slim Bechikh, and **Ali Ouni**, A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection, IEEE Transactions on Software Engineering, 2014.

5. **Ali Ouni**, Marouane Kessentini , Houari Sahraoui and Mohamed Salah Hamdi, Improving Multi-Objective Code-Smells Correction Using Development History. Journal of Systems and Software (JSS), 2014. (under revision).

6. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui, Mel Ó Cinnéide, Kalyanmoy Deb, Automated Multi-Objective Refactoring to Introduce Design Patterns and Fix Anti-Patterns. Journal of Automated Software Engineering, 2014. (under submission).

Publications

7. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui, Multi-Objective Optimization for Software Refactoring and Evolution, Elsevier, Advances in Computers, volume 94, pp. 103-167, 2014.

8. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and Mounir Boukadoum, Maintainability Defects Detection and Correction: A Multi-Objective Approach, in Journal of Automated Software Engineering (JASE), 20(1), pp. 47-79, Springer, 2012.

Refereed Conferences

1. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and M. S. Hamdi, The Use of Development History in Software Refactoring Using a Multi-Objective Evolutionary Algorithm, in the Genetic and Evolutionay Computation Conference (GECCO), pp. 1461-1468, July 2013, Amesterdam, The Netherlands,

2. **Ali Ouni**, Marouane Kessentini and Houari Sahraoui, Search-based Refactoring Using Recorded Code Changes, in the 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 221-230, March 2013, Genova, Italy.

3. **Ali Ouni**, Marouane Kessentini, Houari Sahraoui and M. S. Hamdi, Search-based Refactoring: Towards Semantics Preservation. 28th IEEE International Conference on Software Maintenance (ICSM), pp. 347-356, September 2012, Riva del Garda- Italy.

4. Marouane Kessentini, Wael Kessentini, Houari Sahraoui, Mounir Boukadoum, and **Ali Ouni**, Design Defects Detection and Correction by Example. 19th IEEE International Conference on Program Comprehension (ICPC), pp. 81-90, 22-24 June 2011, Kingston- Canada.

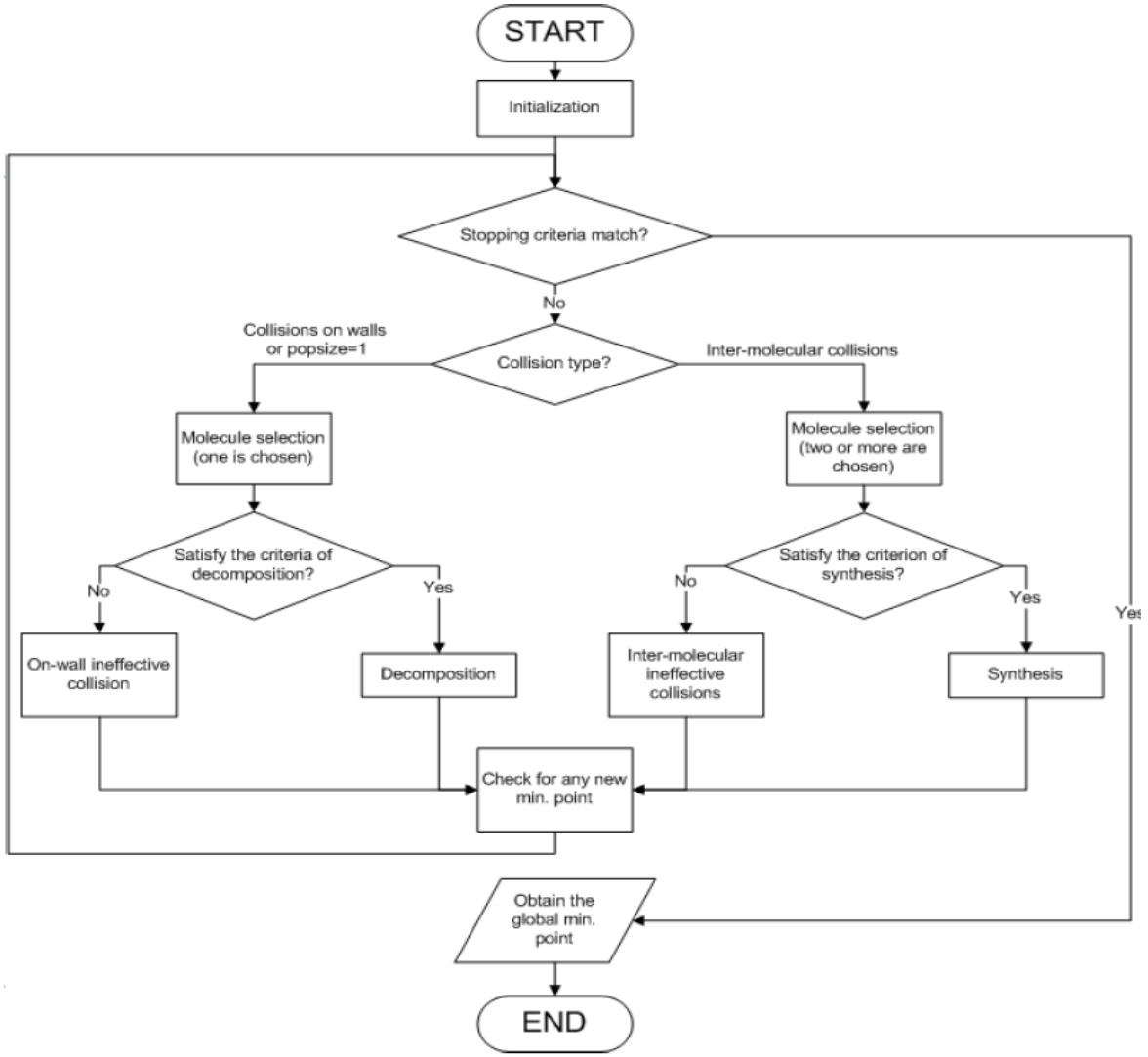
Thank you for your attention

Search-based software engineering

- By Harman and Jones, in 2001
- Formulate software engineering problems as search problems
- Application of optimization techniques
  - Genetic algorithm
  - Simulated Annealing
  - NSGA-II
  - …
- SBSE has become a growing research and practice domain
  - Software testing
  - Requirements engineering
  - Model-driven engineering
  - Project management
  - …

Chemical Reaction Optimisation (CRO)

- Big picture

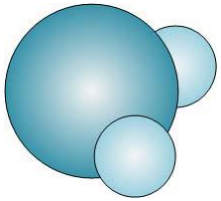


Design semantics

- Semantics
  - Minimize semantic errors
  - The correctness of proposed refactorings increase when applied to semantically connected elements
- Semantic constraints
  - Vocabulary-based similarity (VS)
  - Dependency similarity (DS)
  - Implementation similarity (IS)
  - Feature inheritance usefulness (FIU)
  - Cohesion-based dependency (CD)

Chemical Reaction Optimisation (CRO)

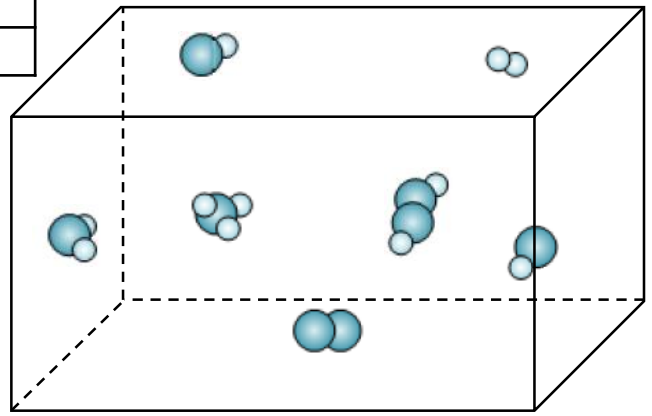
- CRO mimics the interactions of molecules in a chemical reaction to search for the global optimum



- Manipulated agents: Molecule
  - Energy move
- Analogy

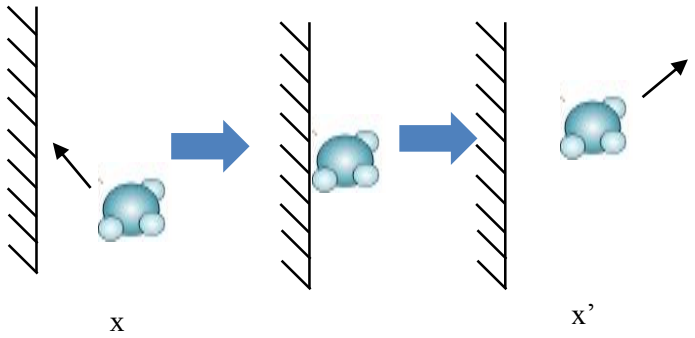
Chemical meaning	Metaheuristic meaning
Molecule	Solution
Potential energy	Objective function value
Collision	Change operator
Kinetic energy	Measure of tolerance of having worse solutions
Number of Hits	Current total number of moves
Minimum structure	Current optimal solution

- A container of molecules

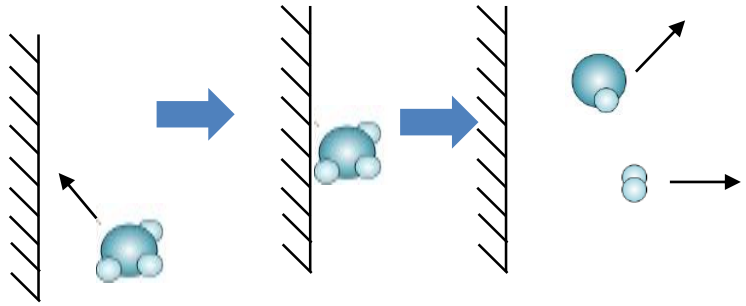


Chemical Reaction Optimisation (CRO)

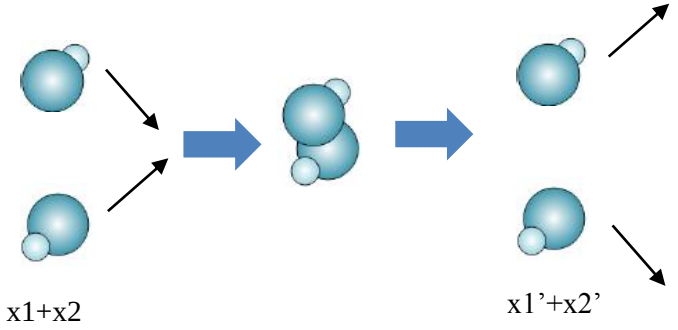
- A series of events take place
  - On-wall ineffective collision



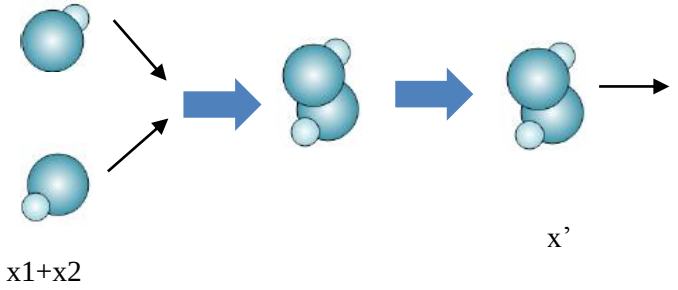
  - Decomposition

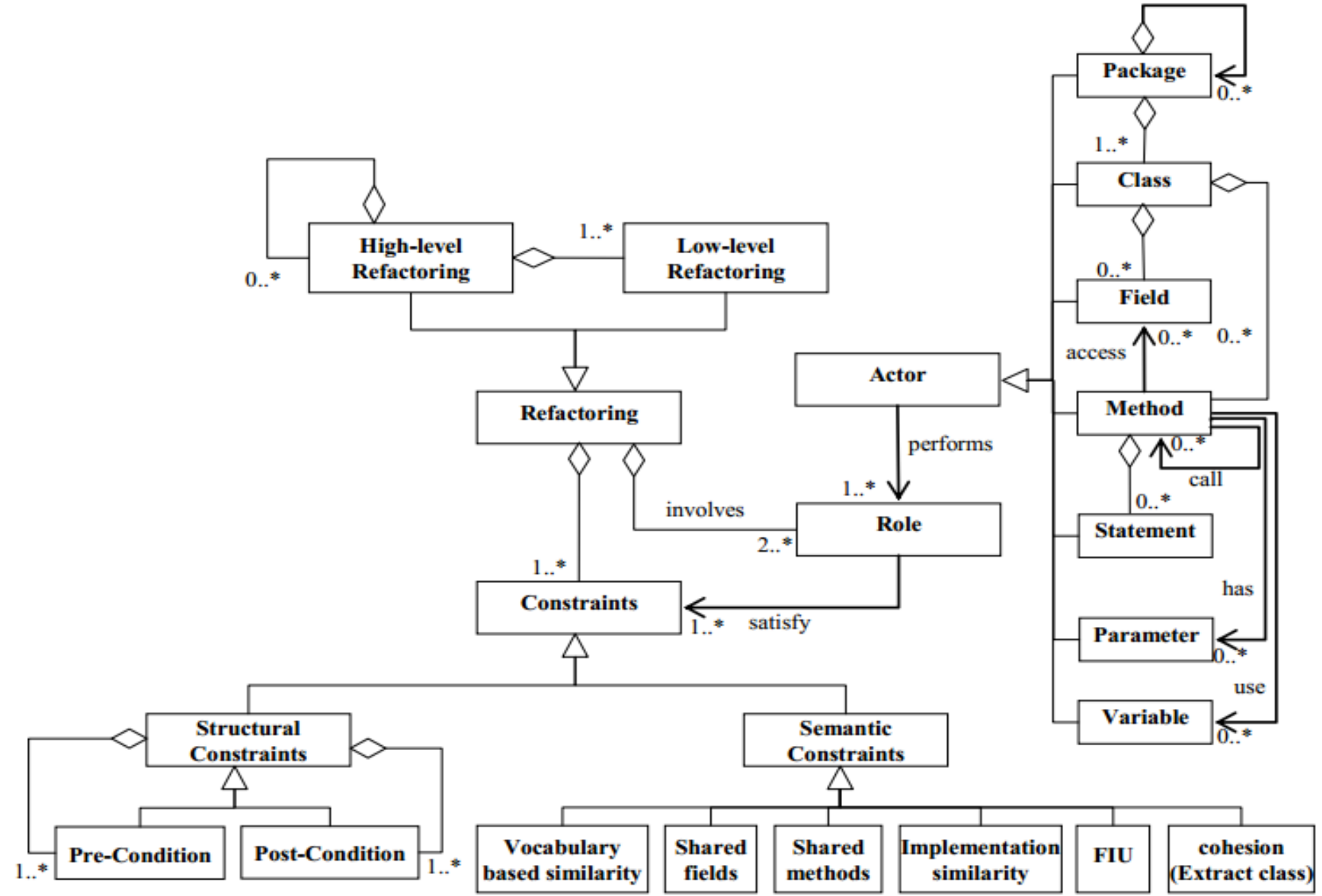


  - Inter-molecular ineffective collision



  - Synthesis




Package
0..*
1..*
Class
0..*
Field
access
0..*
0..*
Actor
Method
performs
0..*
call
1..*
0..*
involves
Role
Statement
2..*
Refactoring
1..*
has
Constraints
1..*
satisfy
Parameter
0..*
use
Variable
0..*
High-level Refactoring
0..*
1..*
Low-level Refactoring
Structural Constraints
Semantic Constraints
Pre-Condition
Post-Condition
1..*
1..*
Vocabulary based similarity
Shared fields
Shared methods
Implementation similarity
FIU
cohesion (Extract class)