Playing with Refactoring

Identifying Extract Class Opportunities through Game Theory

Gabriele Bavota, Rocco Oliveto, Andrea De Lucia
DMI, University of Salerno, Italy

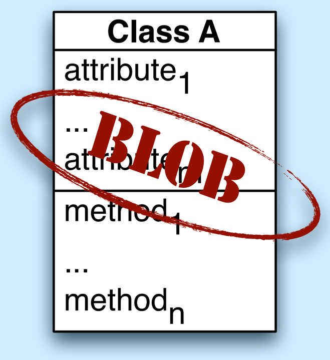
Giuliano Antoniol, Yann-Gaël Guéhéneuc
DGIGL, École Polytechnique de Montreál, Canada

Context

Refactoring Software Systems

Refactoring: changing software without modifying its external behavior

Improving non-functional attribute of the software



Software evolution ... continuos changes

Changes cause a drift of the original design reducing its quality

Class Cohesion: how strongly related the various responsibilities of a class are

Programmers often add wrong responsibilities to a class ⇒ its cohesion decreases

EXTRACT CLASS REFACTORING Splitting a class with many responsibilities into different classes

Game Theory Background

The Prisoner's Dilemma

Game Theory: capture behavior in strategic situations, in which an individual's success when making choices depends on the choices of others

A **game** consists of
- a set of players (2 or more)
- a set of moves available to those players
- payoffs for each combination of moves

The Prisoner's Dilemma

Sally and Tom are accused of fraudulent activity and both want to minimize the time spent in jail

The solution of this game is represented by the **Nash equilibrium** (confess, confess)

		Tom: confess	Tom: not confess
Sally	confess	**(5, 5)**	(0, 7)
	not confess	(7, 0)	(4, 4)

Payoff matrix for the Prisoner's Dilemma

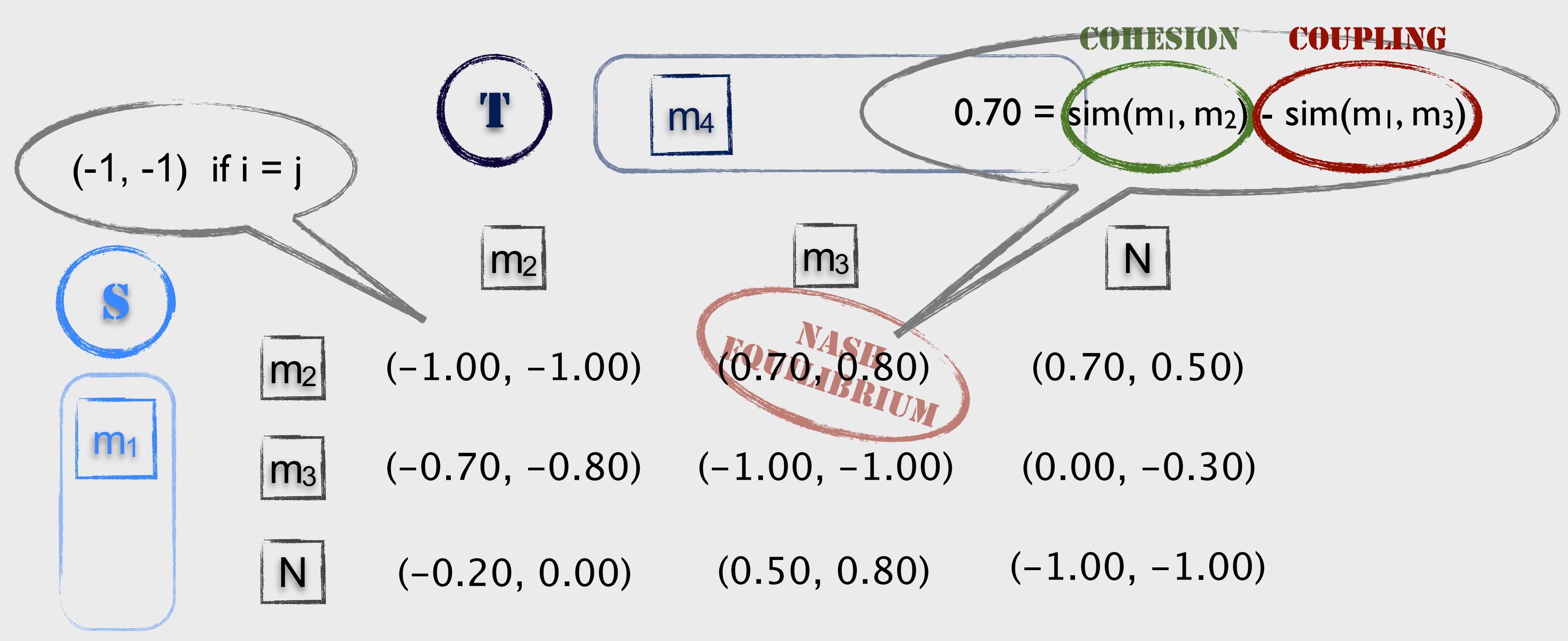
Given the non-cooperative nature of this game the minimum sentence for both players can be obtained only if both the players confess

Game Theory Meets Software Engineering

Modeling Extract Class Refactoring as a Non-cooperative Game

2 Players	Each player is in charge to build a new class selecting methods from the original class
Moves	Iterative game: at each iteration, a player selects at most one method of the class to be refactored
Payoff	Each player selects the method considering the impact on the cohesion and coupling of his class

The game starts by assigning to *S* and *T* the two methods having the lowest similarity, e.g., m1, and m4. The similarity between two methods, i.e, *sim*, is obtained as a combination of structural and semantic metrics.
The move "N" represents the *null move*: a player that selects this move during an iteration doesn't take any method. In this way we avoid the trivial splitting of a class in two classes of the same dimension and increase the rationality of the players.



The move to be performed during an iteration of the process is chosen by finding the ***Nash equilibrium*** in the payoff matrix

Preliminary Evaluation

Case Study Design

	Goal	Systems
RQ1	Comparison with Pareto optimum	ArgoUML, JHotDraw
RQ2	Comparison with others extract class techniques	ArgoUML, JHotDraw

Experiment execution

The evaluation planning is inspired by **mutation testing**: we randomly select two classes of one of the object systems, merge them in a single class Cm and then use the experimented approaches to split the merged class in two classes

Results (F-Measure)

System	Game Theory	Pareto Optimum	MaxFlow MinCut
ArgoUML	90%	88%	77%
JHotDraw	85%	82%	76%

Reconstruction accuracy

26th IEEE International Conference on Software Maintenance, Timişoara, Romania, September 15, 2010