Evaluating Design Decay during Software Evolution

Salima Hassaine

Under the supervision of:
Yann-Gaël Guéhéneuc[1] and Sylvie Hamel[2]
[1] Ptidej Team – DGIGL, École Polytechnique de Montréal, Canada
[2] LBIT – DIRO, Université de Montréal, Canada

Ph.D. Defense
December 17^{th}, 2012

Université de Montréal ÉCOLE POLYTECHNIQUE MONTRÉAL Ptidej

Pattern Trace Identification, Detection, and Enhancement in Java

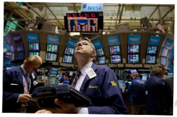
Outline

Context and Motivation (1/7)

- Software systems play a crucial role in modern societies. They are everywhere from small game applications to large embedded systems
- Software developers build larger and more complex software

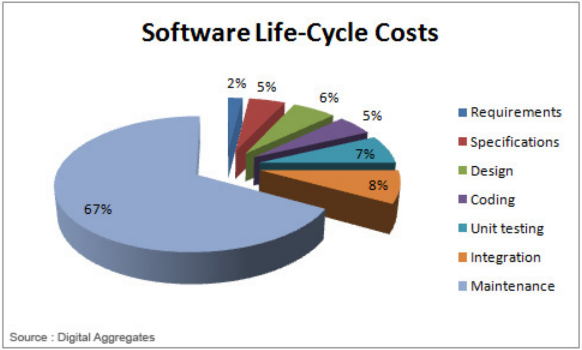







Context and Motivation (2/7)

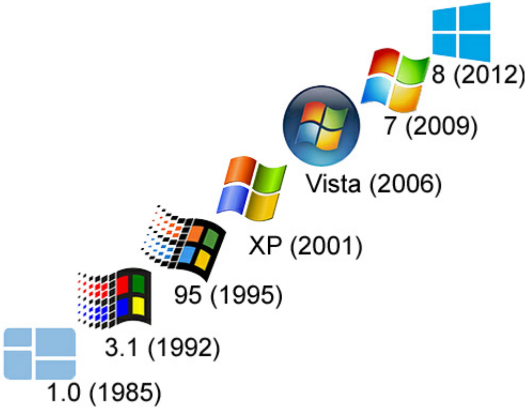
- Software maintenance is the most costly and difficult activity [1]
- The maintenance effort has been estimated to be more than 70% of the overall software development cost [1]



[1] Ian Sommerville. *Software Engineering*. Addison-Wesley, 6^{th} edition, 2000.

Context and Motivation (3/7)

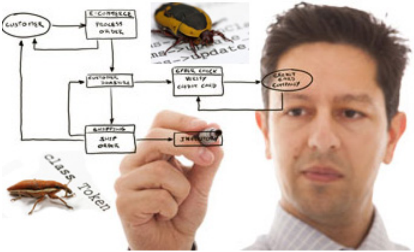
- Software systems evolve continuously, requiring continuous maintenance and development [2]



[2] M. M. Lehman. *Laws of Software Evolution Revisited*. In Proceedings of the 5th European Workshop on Software Process Technology, 1996.

Context and Motivation (4/7)

- Software design tends to decay with time and becomes less adaptable to new requirements [3,4]
- Design decay occurs when developers do not understand the original design [4]



[3] Jilles van Gurp and Jan Bosch. *Design erosion: problems and causes*. Journal of Systems and Software, 61(2): 105-119, 2002.

[4] David L. Parnas. *Software aging*. In Proceedings of the 16^{th} International Conference on Software Engineering, ICSE'94, 279-287, 1994.

Context and Motivation (5/7)

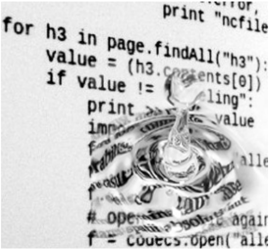
- Future changes become more difficult and are more likely to introduce new bugs [4]
- Experience shows that 40% of bugs are introduced while correcting previous bugs [5]

[4] David L. Parnas. *Software aging*. In Proceedings of the 16th International Conference on Software Engineering, ICSE'94, 279-287, 1994.

[5] R. Purushothaman and D. E. Perry. *Toward understanding the rhetoric of small source code changes*. IEEE Transactions on Software Engineering, 2005.

Context and Motivation (6/7)

- Making changes without understanding their effects may lead to the **introduction of bugs** [6]
- Understanding **change propagation** requires source code analysis, which is a difficult and error-prone activity [7]



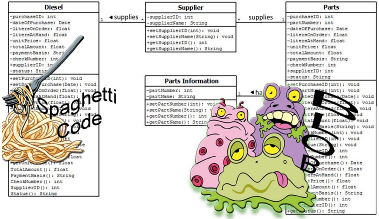
[6] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996

[7] S. Pfleeger, *Software Engineering: Theory and Practice*. PrenticeHall, 1998

Context and Motivation (7/7)

- Developers who lack knowledge and experience may introduce **design defects** [8]

- Developers **spend a lot of time in correcting defects** before completing a maintenance task [9]



[8] W. J. Brown *et al.* *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, 1998

[9] Z. Xing. *Analyzing the evolutionary history of the logical design of object-oriented software*. IEEE Transactions on Software Engineering, 2005

Motivating Example (1/2)

- *On 1998, Netscape decided to release their own browser as open source. After 6 months, the developers decided to start rewriting another version from scratch* [3]
- *AOL announced that on February* 1st*, 2008 it would drop support for the Netscape web browser and would no longer develop new releases* [10]

[3] Jilles van Gurp and Jan Bosch. *Design erosion: problems and causes*. Journal of Systems and Software, 61(2): 105-119, 2002.

[10] `http://blog.netscape.com`

Motivating Example (2/2)

- It's a large project, and it **takes a long time for a new developer to dive in** and start contributing [11]
- The code was too hard to modify ... when developers **try to make a small change** and find that **it's taking them longer** than few hours, **they give up** [11]

[11] Web page of Jamie Zawinski: `http://www.jwz.org/gruntle/nomo.html`

Thesis

Software maintenance is severally impacted by **design decay, uncontrolled changes, and design defects**. *Therefore, to assist developers during software maintenance, we propose to* **evaluate design decay**, *to* **analyse change impact**, *and to* **detect design defects**.

Contributions

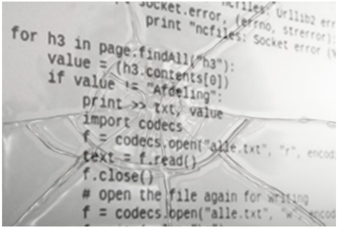
(1) **Design Decay Evaluation**.
Developers should detect classes that are decaying. These classes should be fixed to control their decay

(2) **Change Impact Analysis**.
Once developers decide which classes should be fixed they can analyse the impact of their changes

(3) **Design Defects Detection**
Finally, developers should improve the quality of software design by detecting design defects

Design Decay

- "**Design Decay** *is the deviation of actual or concrete design from planned or conceptual design*" [4]
- "**Design Decay** *is the cumulative, negative effect of changes on the quality of a software system*" [12]

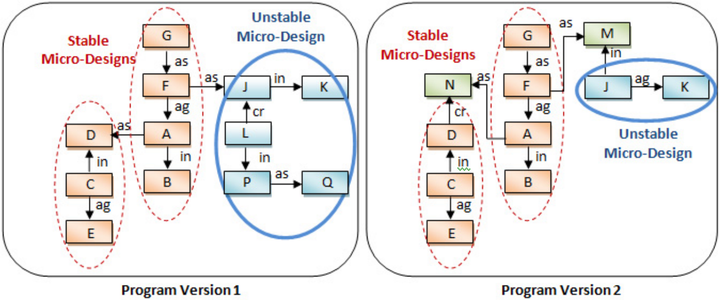


[4] David L. Parnas. *Software aging*. In Proceedings of the 16^{th} International Conference on Software Engineering, ICSE'94, 279-287, 1994

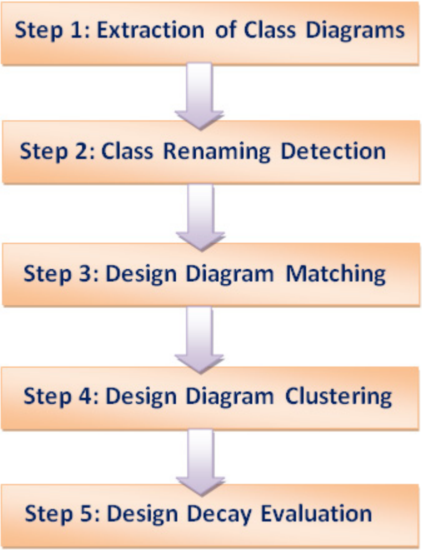
[12] Van Gurp *et al.* *Design preservation over subsequent releases of a software product: a case study of Baan ERP*. In Journal of Software Maintenance and Evolution, 277-306, 2005

Our goal

- Identification of structural changes that invalidate the original design
- Identification of stable and unstable of micro-designs
- Evaluation of design decay



Approach ADvISE



Step 1: Extraction of Class Diagrams

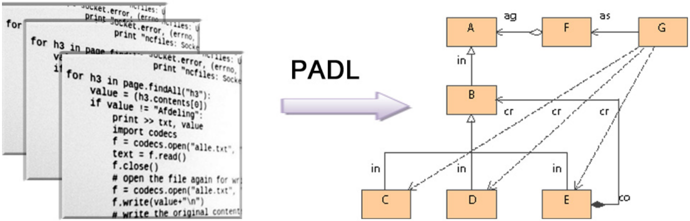


Figure: An example of class diagram (PADL Model [13])

[13] Yann-Gaël Guéhéneuc and Giuliano Antoniol. *DeMIMA: A Multi-layered Framework for Design Pattern Identification*. IEEE Transactions on Software Engineering, 2008

Step 2: Class Renaming Detection (1/4)

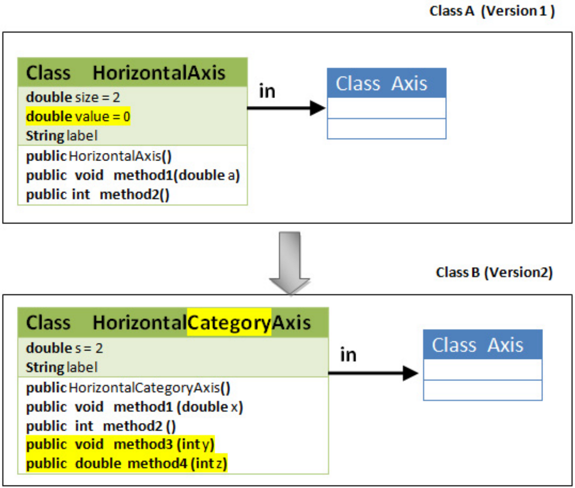


Figure: Example of class renaming

Step 2: Class Renaming Detection (2/4)

(1) Structural Similarity:

$$StrS(C_A, C_B) = \frac{2 \times |S(C_A) \cap S(C_B)|}{|S(C_A)| + |S(C_B)|} \in [0, 1]$$

Example 1:
$S(C_A) \cap S(C_B) = \{$2 attribute types (`String` and `double`), 1 constructor, 2 methods (`void method1(double)` and `int method2()`), 1 inheritance $\}$.
$|S(C_A) \cap S(C_B)| = 6$, $|S(C_A)| = 9$, $|S(C_B)| = 6$.

$$StrS(C_A, C_B) = \frac{2 \times 6}{9 + 6} = 0.80$$

Step 2: Class Renaming Detection (3/4)

(2) Camel Similarity

$$CamelS(C_A, C_B) = \frac{2 \times |T(C_A) \cap T(C_B)|}{|T(C_A)| + |T(C_B)|} \in [0, 1]$$

Example 2:

$T(C_A) = \{\text{Horizontal, Axis}\}$
$T(C_B) = \{\text{Horizontal, Category, Axis}\}$.
$|T(C_A)| = 2$, $|T(C_B)| = 3$, $|T(C_A) \cap T(C_B)| = 2$.

$$CamelS(C_A, C_B) = \frac{2 \times 2}{2 + 3} = 0.8$$

Step 2: Class Renaming Detection (4/4)

(3) Normal Edit Distance

$$ND(C_A, C_B) = \frac{LEV(C_A, C_B)}{length(C_A) + length(C_B)} \in [0, 1]$$

Example 3:

$$ND(C_A, C_B) = \frac{8}{14 + 22} = 0.22$$

.

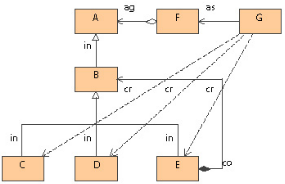
(4) Combination of all similarities:

(1) $StrS(C_A, C_B) \nearrow$, $CamelS(C_A, C_B) \nearrow$, $ND(C_A, C_B) \searrow$

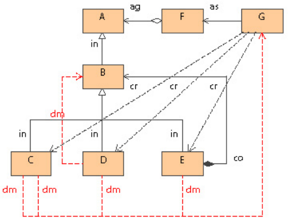
(2) $CamelS(C_A, C_B) \geq 0.5$, $ND(C_A, C_B) \leq 0.4$

Step 3: Design Diagram Matching (1/2)

Generation of the String Representation (EPI tool [14])



(a) Class Diagram



(b) Eulerian Model

A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A

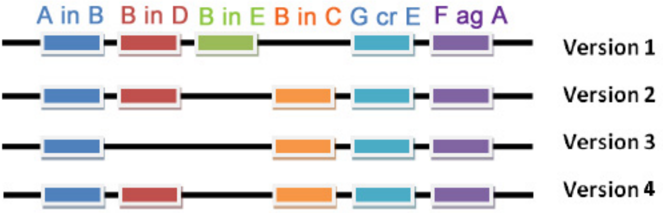
(c) Generating the string representation

[14] O. Kaczor, Y.-G. Guéhéneuc, and S. Hamel, *Efficient identification of design patterns with bit-vector algorithm*, In Proceedings of the 10^{th} European Conference on Software Maintenance and Reengineering, pp.175–184, 2006.

Step 3: Design Diagram Matching (2/2)

Bit-Vector Algorithm

- Input: List of class renamings and string representations of program versions
- Output: Sets of triplets stables/unstables



Step 4: Design Diagram Clustering

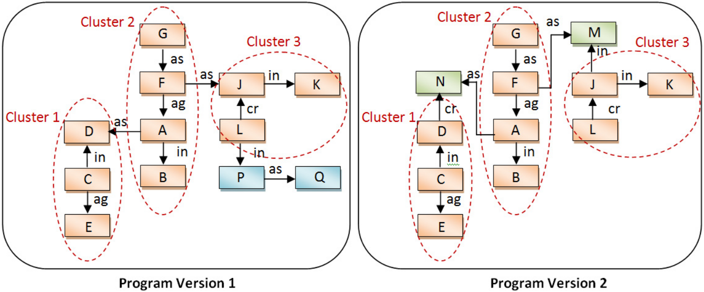


Figure: Example of Clustering, each Cluster represents a $S\mu_D$

Step 5: Design Decay Evaluation

- **Tunnel Triplets Metric (TTM(i))**

$$S_{Tunnel}(i) = \{T \in Triplets | T \in V_j, \forall j \in [0, i]\}\}$$

$$TTM(i) = |S_{Tunnel(i)}|$$

- **Common Triplets Metric (CTM(i,j))**

$$ST(i,j) = \{T \in Triplets | T \in V_n, \forall n \in [k,j], \exists k \in [i,j[\}$$

$$CTM(i,j) = |ST(i,j)|$$

- where *Triplets* is the set of all triplets $T = (C_{Source}, R, C_{Target})$.

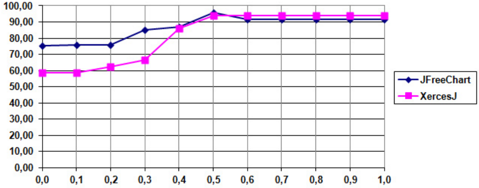
Empirical Study Design

System	Releases	Entities (in classes)	Bit-vectors (in bits)	History (in releases)
ArgoUML	v0.10.1 v0.34	1447 1984	12,265,560 105,456,260	17
DNSjava	v1.2.0 v2.1.3	164 124	49,759 93,067	33
JFreeChart	v0.5.6 v1.0.13	100 778	87,227 1,089,345	51
Rhino	v1.5.R1 v1.6.R5	163 449	40,803 266,265	11
XercesJ	v1.0. v2.9.0	296 697	162,583 1,195,353	36

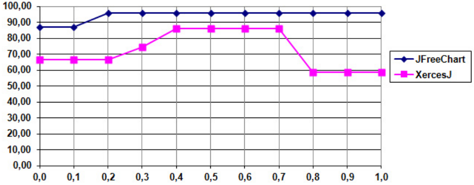
Table: Statistics for the first and last version of each system

Results (1/6)

RQ1: What are the thresholds for class renaming detection?



(a) F-measure (Camel Similarity)



(b) F-measure (Normalized Edit Distance)

Results (2/6)

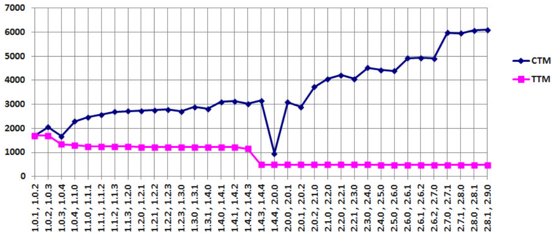
RQ2: What is the efficiency of ADvISE for class renaming detection in a software system?

Systems	Similarities	CamelS	ND	StrS	Combination
JFreeChart v0.5.6-v1.0.13	Precision	65.90%	77.27%	72.72%	95.45%
	Recall	67.41%	79.06%	74.41%	97.67%
XercesJ v1.0.1-v2.9.0	Precision	84.61%	38.46%	57.69%	92.30%
	Recall	88.00%	40.00%	60.00%	96.00%

Results (3/6)

RQ3: What are signs of design decay and how can they be tracked down?

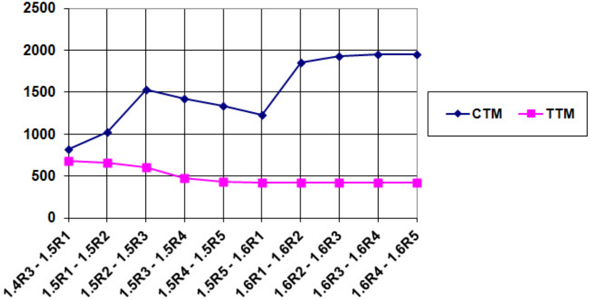
- **XercesJ 1.4.4 – 2.0.0**: "XercesJ 2.0.0 is a nearly complete rewrite of the XercesJ 1.x code base to make the code cleaner, more modular, and easier to maintain. It includes a completely redesigned and rewritten XML Schema validation engine"



Results (4/6)

RQ3: What are signs of design decay and how can they be tracked down?

- **Rhino 1.5R5 – 1.6R1**: *Rhino 1.6R1 as the new major release of Rhino*, there are important changes in Rhino 1.6R1, "... *without affecting the existing code base*"



Results (5/6)

RQ4: Do stable and decaying micro-designs have the same bug-proneness?

	Bug-prone classes	Clean classes
$\mathbf{D}\mu_{\mathbf{A}}$	973	763
$\mathbf{S}\mu_{\mathbf{A}}$	148	301
Fisher's test $(\mathbf{p}-\mathbf{value})$	$2.2e^{-16}$	
Odd-ratio (OR)	2.59	

RQ5: Do stable and decaying micro-designs have the same design defect-proneness?

	Design defect-prone classes	Clean classes
$\mathbf{D}\mu_{\mathbf{D}}$	1305	431
$\mathbf{S}\mu_{\mathbf{D}}$	210	239
Fisher's test $(\mathbf{p}-\mathbf{value})$	$2.2e^{-16}$	
Odd-ratio (OR)	3.44	

Table: Contingency tables (ArgoUML) and Fisher's test

Results (6/6)

RQ6: How effective is ADvISE?

	Pre-processing		ADvISE			
Systems	**PADL**	**EPI**	**Step 2**	**Step 3**	**Step 4**	**Step 5**
ArgoUML	7.047	18,098.000	4.835	10.651	10.140	908.329
DNSjava	2.249	44.209	0.862	0.935	0.075	7.150
JFreeChart	2.197	62.268	3.135	1.907	0.099	50.030
Rhino	2.150	50.350	1.783	0.450	0.064	7.985
XercesJ	4.520	179.410	1.273	0.549	0.032	15.488
Median	2.249	62.268	1.783	0.935	0.075	15.488
Average	3.632	3,686.840	2.377	2.898	2.082	197.796

Table: Execution time (in seconds) for each step of ADvISE

Lessons learned ...

- Our metrics provide valuable insight about design decay
  - If TTM decreased, then the original design decayed
  - If TTM is stable, then the original design is stable
  - If CTM increased, then there are new requirements
  - If CTM is stable, then the system is stable and the most of maintenance activities are bug fixes
- **Decaying classes** are more **bug-prone** and **defect-prone** than stable classes
- Class renamings detection has good precision/recall
- Design diagram matching using Bit-vector is efficient

Contributions

(1) **Design Decay Evaluation**
Developers should detect classes that are decaying. These classes should be fixed to control their decay

(2) **Change Impact Analysis**
Once developers decide which classes should be fixed they can analyse the impact of their changes

(3) **Design Defects Detection**
Finally, developers should improve the quality of software design by detecting design defects

Change Impact Analysis

- **Change impact analysis** is defined by Bohner and Arnold [15] as "*identifying the potential consequences of a change, or estimating what needs to be modified to accomplish a change*".

[15] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

Existing approaches (1/3)

- **Structure-based Analysis**
  - Dependency analysis of source code is performed using static or dynamic program analyses
  - The relationships between classes make change impact difficult to anticipate (*e.g.*, hidden propagation)

[15] S. A. Bohner and R. S. Arnold, *Software Change Impact Analysis*. IEEE Computer Society Press, 1996

[16] J. Law *et al.*, *Whole program Path-Based dynamic impact analysis*. In proceedings of ICSE, 2003

[17] X. Zhang *et al.*, *A study of effectiveness of dynamic slicing in locating real faults*. Journal of Empirical Software Engineering, 2007

Existing approaches (2/3)

- **History-based Analysis**
  - Mining software repositories to identify co-changes of software artefacts within a change-set
  - It is often able to capture change couplings that cannot be captured by static and dynamic analyses
  - They lack to capture how changes are spread over space (*e.g.*, class diagram) ⇛ They could not help developers prioritise their changes according to the **forecast scope** of changes

[18] T. Zimmermann *et al.*, *Mining Version Histories to Guide Software Changes*. In proceedings of ICSE, 2004

[19] Annie T. Ying *et al.*, *Source code that talks: an exploration of Eclipse task comments and their implication to repository mining*. MSR, 2005

[20] S. Bouktif *et al.*, *Extracting Change-patterns from CVS Repositories*. In proceedings of WCRE, 2006

Existing approaches (3/3)

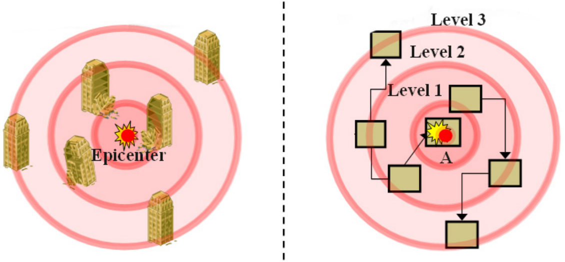
- **Probabilistic Approaches**
  - Building change propagation models to predict future change couplings using probabilistic tools (*e.g.*, Bayesian Networks, Time Series Analysis, etc.)
  - They lack to capture how changes are spread over space (*e.g.*, class diagram) $\Rrightarrow$ They could not help developers prioritise their changes according to the **forecast scope** of changes

[21] S. Mirarab *et al.*, *Using Bayesian Belief Networks to Predict Change Propagation in Software Systems*. In Proceedings of ICPC, 2007

[22] Y. Zhou *et al.*, *A Bayesian Network Based Approach for Change Coupling Prediction*. In Proceedings of WCRE, 2008

[23] M. Ceccarelli *et al.*, *An eclectic approach for change impact analysis*. In Proceedings of ICSE, 2010

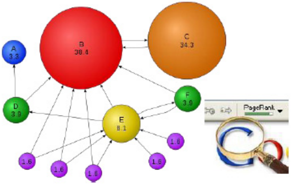
Approach: Seismology-inspired Metaphor



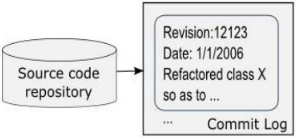
Active seismic areas	"Important" classes
Earthquake	Software change
Epicenter	"Important" changed class
Seismic wave propagation	Change propagation
Damaged sites	"Impacted" classes
Distance from an epicenter	Class level

Step 1: Identifying the most important classes

PageRank-based Metric



History-based Metric



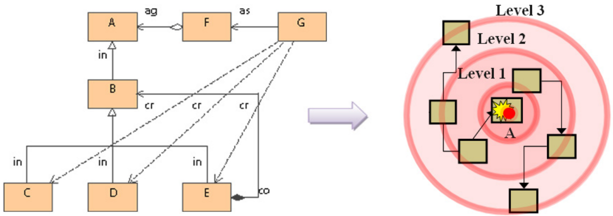
Combination

$$rh(c) = \frac{r(c)}{h(c)}$$

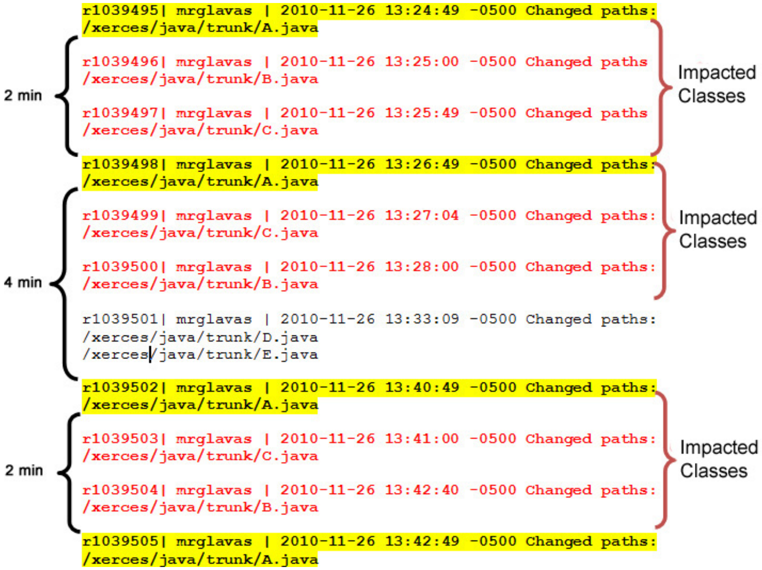
Step 2: Identifying class levels

Bit-Vector Algorithm

- Input:
  - The Epicenter Class (*e.g.*, **class** A)
  - The String Representation of the program
- Output:
  - Class levels (*e.g.*, Level0 $= \{A\}$, Level1 $= \{B, F\}$, Level2 $=\{D, E, C\}$, Level3 $= \{G\}$)

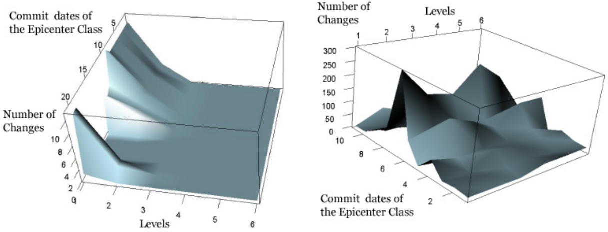


Step 3: Identifying impacted classes



Empirical Study Results (1/2)

RQ1: Does our metaphor allow us to observe the scope of change propagation?



(e) class XMLEventImpl (f) class TypeValidator

Figure: Change propagation

- Epicenter class `XMLEntityScanner`: we found the bug ID1099 that relate the changes to the epicenter class with changes to `XMLParser` (level 3).

Empirical Study Results (2/2)

RQ2: What is the level most impacted by a change?

	Homogenous subsets for alpha = 0.1		
Levels	Range 1	Range 2	Range 3
6	6.4015		
5	10.8485		
4	24.8333		
3		50.2789	
2		83.7273	
1			895.2652

Table: Xerces-J: Duncan's test applied on "number of changes"

RQ3: What is the farthest reached level by a change?

	Homogenous subsets for alpha = 0.1		
Max Level	Range 1	Range 2	Range 3
6	10.5333		
5	16.3333		
4	21.6667		
3		30.0033	
2		43.2000	
1			54.8667

Lessons learned ...

- Seismology provides an interesting metaphor for identifying the scope of change propagation
- The scope of change propagation could reach the 6^{th} level. Thus, our intuition, about the impacted classes by a change must be near to the changed class, is incorrect in some cases
- Identifying the scope of change propagation could help developers to rapidly pinpoint the source of a bug by only analysing the indicated levels in priority instead of inspecting all the source code

Contributions

(1) **Design Decay Evaluation**
Developers should detect classes that are decaying. These classes should be fixed to control their decay

(2) **Change Impact Analysis**
Once developers decide which classes should be fixed they can analyse the impact of their changes

(3) **Design Defects Detection**
Finally, developers should improve the quality of software design by detecting design defects

Design Defects

- **Design Defects** are "*bad solutions to recurring software design and implementation problems. They are conjectured to have a negative impact on the quality and life-time of systems*" [6,8]

[8] W. J. Brown *et al. Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley and Sons, 1^{st} edition, 1998.

[24] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1^{st} edition, 1999.

Existing approaches (1/4)

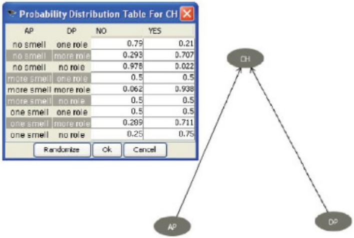
- **DECOR: Rule Cards based on fixed threshold**
  - Cannot report accurate information for borderline classes (Submarine effect)
  - Requires experts'knowledge and interpretation to define the rule cards

```
RULE_CARD : SpaghettiCode {
 ...
 RULE: LongMethodMethodNoParameter {INTER LongMethod MethodNoParameter};
 RULE: LongMethod {(METRIC: LOC_METHOD, VERY_HIGH)};
 RULE: MethodNoParameter {(METRIC: NB_PARAM, 0)};
 ...
 RULE: NoInheritance {(METRIC: DIT, 1)} ;
 RULE: FunctionClassGlobalVariable {UNION FunctionClass GlobalVariable};
 RULE: FunctionClass {(SEMANTIC: CLASSNAME, {Make, Create, Creator, Exec}) };
 RULE: GlobalVariable {(STRUCT: FIELD, CLASS_GLOBAL_VAR)};
```

[25] N. Moha *et al.*, *DECOR: A Method for the Specification and Detection of Code and Design Smells.* In journal of TSE, 2009

Existing approaches (2/4)

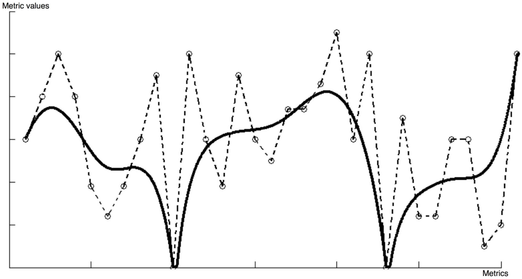
- **BBN: returns the probabilities of classes to be antipatterns but...**
  - Input Nodes: characterizations of the design of a class.
  - Output Nodes: probability that the class is an antipattern
  - Requires experts'knowledge to define a learning structure



[26] F. Khomh *et al.*, *A Bayesian Approach for the Detection of Code and Design Smells*. In Proceedings of QSIC, 2009

Existing approaches (3/4)

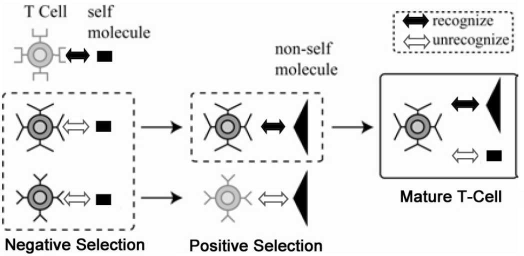
- **ABS (Antipattern identification using B-Splines)**
  - A class is modeled using specific interpolation curves (i.e., B-splines) of plots mapping metrics and their values for the class
  - Focuses on detecting one kind of design smells at a time



[27] R. Oliveto *et al.*, *Numerical Signatures of Antipatterns: An Approach based on B-Splines*. In Proceedings of CSMR, 2010

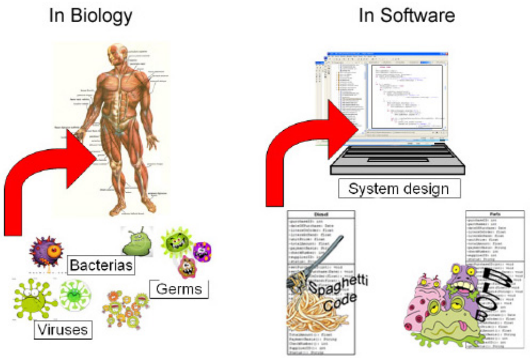
Existing approaches (4/4)

- **Kessentini et al: returns the risk of classes but...**
  - Input: characterizations of a good design...
  - Output: risk that the class is an antipattern
  - There is no guarantee of obtaining the same results for different runs



[28] M. Kessentini *et al.*, *Deviance from perfection is a better criterion than closeness to evil when identifying risky code*. In Proceedings of ASE, 2010

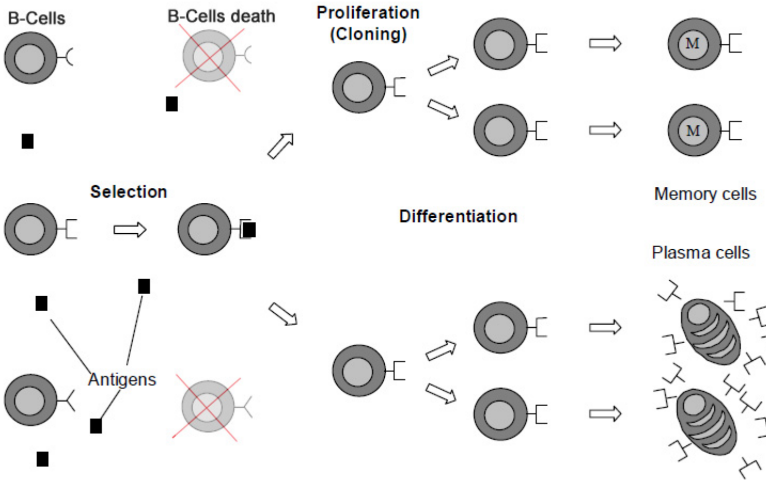
Approach: Immune-inspired Metaphor



Concepts of Immune System	
In Biology	In Software
Body	Software design
Immune system	Design defects detection approach
Antigen	Sequence of quality metrics
Antibody	Known pattern of quality metrics values (Defect Class)
Affinity	Similarity measure between sets of metrics values

Table: Instantiation of an AIS to detect design defects

Clonal Selection Principle in Biology



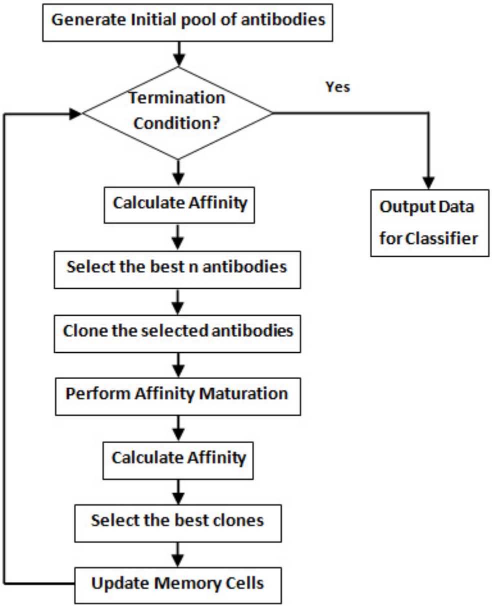
Artificial Immune System

- **Encoding of Antigens and Antibodies**
  Vector $X = \{x_1, x_2, ..., x_n, y\}$, where x_i is a real number representing a quality metric ($x_i \in R$ for $i \in [1..n]$), and $y = \{+1, 1\}$ is a label (defect class or clean class)

- **Affinity Measure (Euclidean Distance (ED))**
  Between an $Antigen{=}(ag_1, ag_2, ..., ag_k)$ and an $Antibody{=}(ab_1, ab_2, ..., ab_k)$, given by

$$ED(Ag, Ab) = \sqrt{\sum_{i=1}^{k}(ag_i - ab_i)^2}$$

CLONALG Algorithm



Empirical Study

	Numbers of				
	Classes	KLOCs	Blobs	FDs	SCs
Gantt Project	188	31	4	4	4
XercesJ	589	240	15	15	18
Total	777	271	19	19	22

Table: System characteristics

Empirical Study Results (1/3)

RQ1: To what extent an AIS-based approach can detect design defects in a system?

	Numbers of			
	Design Defects	False Positives	Precisions	Recalls
Subset 1	16	1	94.11%	100%
Subset 2	16	2	88.23%	100%
Subset 3	16	2	88.23%	100%
Average			90.19%	100%

Table: Intra-system detection on XercesJ: 3-fold cross validation

	Numbers of			
	Design Defects	False Positives	Precisions	Recalls
GanttProject (on XercesJ)	20	7	65.0%	100%
XercesJ (on GanttProject)	54	10	81.48%	100%

Table: Inter-system detection, trained on Blobs, FDs and SCs

Empirical Study Results (2/3)

RQ2: Is our approach better than state of-the-art approaches, such as DECOR and BBNs?

	GanttProject	XercesJ
DECORE	26.73%	36.22%
BBN	57.10%	36.50%
IDS	65.00%	81.48%

Table: Results of comparing the detection approaches

Lessons learned ...

- The immune system provides an interesting metaphor for detecting design defects
- The CLONALG algorithm provides good performance in time, precision, and recall
- The CLONALG algorithm detects design defects in general: although we train our approach on only three kinds of design defects, it can detect any kind of design defects

Conclusion

- Stable designs are easier to implement, change, and maintain
- Decaying classes are more bug-prone and defect-prone than stable classes
- The detection of decaying designs early in the process substantially reduce the cost of subsequent steps of software development
- Design decay is inevitable, but it can be slow down if we control software changes and software quality

Future work (Short Term)

Design Decay Evaluation

- Analysing class renamings
- Investigating other metrics to estimate the "mortality" rate of classes

Change Impact Analysis

- Applying our approach on other systems to compute its precision and recall

Artificial Immune Systems

- Comparing our approach with other machine learning techniques, such as support vector machine, and to further study the parameters of the approach, including refining the choice of characteristics of classes

Future work (Long Term)

Design Decay Evaluation

- Identifying refactoring opportunities to fix decaying designs

Change Impact Analysis

- Predicting futures changes using seismology metaphore
- Studying the type of changes which favors change propagation

Artificial Immune Systems

- Predicting changes which lead to the introduction of bugs

Publications

Journal Papers

1. **Salima Hassaine**, Fehmi Jaafar, Yann-Gäel Guéhéneuc, Sylvie Hamel and Bram Adams (submitted on 2012). Evaluating Design Decay during Software Evolution, Journal of Empirical Software Engineering (EMSE), 36 pages

Conference Papers

1. Fehmi Jaafar, **Salima Hassaine**, Yann-Gäel Guéhéneuc, Sylvie Hamel and Bram Adams (2013). **Program Evolution and Bug-proneness: An Empirical Study**. CSMR'13.
2. **Salima Hassaine**, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giulio Antoniol (2012). **ADvISE: Architectural Decay In Software Evolution**. CSMR'12.
3. **Salima Hassaine**, Ferdaous Boughanmi, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giulio Antoniol (2011). **A Seismology-inspired Approach for Change Impact Analysis**. ICSM'11.
4. **Salima Hassaine**, Ferdaous Boughanmi, Yann-Gäel Guéhéneuc, and Sylvie Hamel and Giuliano Antoniol (2011). **Change Impact Analysis : An earthquake Metaphor**. ICPC'11.
5. **Salima Hassaine**, Foutse Khomh, Yann-Gaël Guéhéneuc, and Sylvie Hamel (2010). **IDS: An Immunology-inspired Approach for the Detection of Software Design Smells**. QUATIC'10.

Thesis

Software maintenance is severally impacted by **design decay, uncontrolled changes, and design defects**. *Therefore, to assist developers during software maintenance, we propose to* **evaluate design decay**, *to* **analyse change impact**, *and to* **detect design defects**.