Evaluating Design Decay during Software Evolution

Salima Hassaine

Context and Motivation
Evaluating Design Decay
Change Impact Analysis
Design Defects Detection
Conclusion and Future work

Salima Hassaine

Under the supervision of:
Yann-Gaël Guéhéneuc¹ and Sylvie Hamel²

¹ Ptidej Team – DGIL, École Polytechnique de Montréal, Canada
² LBIT – DIRO, Université de Montréal, Canada

Ph.D. Defense
December 17th, 2012
Outline

1. Context and Motivation
2. Evaluating Design Decay
3. Change Impact Analysis
4. Design Defects Detection
5. Conclusion and Future work
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
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]

Context and Motivation (3/7)

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

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]


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]


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]


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]


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]


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]

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]


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

Step 2: Class Renaming Detection

Step 3: Design Diagram Matching

Step 4: Design Diagram Clustering

Step 5: Design Decay Evaluation

Figure: Approach Overview
Step 1: Extraction of Class Diagrams

![Class Diagram Example](image)

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

Step 2: Class Renaming Detection (1/4)

Figure: Example of class renaming
Step 2: Class Renaming Detection (2/4)

(1) Structural Similarity:

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

Example 1:
\[ S(C_A) \cap S(C_B) = \{ \text{2 attribute types (String and double), 1 constructor, 2 methods (void method1(double) and int method2()), 1 inheritance} \} \]
\[ |S(C_A) \cap S(C_B)| = 6, \ |S(C_A)| = 9, \ |S(C_B)| = 6. \]
\[ \text{StrS}(C_A, C_B) = \frac{2 \times 6}{9 + 6} = 0.80 \]
Step 2: Class Renaming Detection (3/4)

(2) Camel Similarity

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

Example 2:
\[ T(C_A) = \{ \text{Horizontal, Axis} \} \]
\[ T(C_B) = \{ \text{Horizontal, Category, Axis} \}. \]
\[ |T(C_A)| = 2, |T(C_B)| = 3, |T(C_A) \cap T(C_B)| = 2. \]

\[ \text{CamelS}(C_A, C_B) = \frac{2 \times 2}{2 + 3} = 0.8 \]
Step 2: Class Renaming Detection (4/4)

(3) Normal Edit Distance

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

Example 3:

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

(4) Combination of all similarities:

(1) \( StrS(C_A, C_B) \uparrow, CamelS(C_A, C_B) \uparrow, ND(C_A, C_B) \downarrow \)

(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

(c) Generating the string representation

\[ A \text{ in } B \text{ in } D \text{ dm } B \text{ in } E \text{ co } B \text{ in } C \text{ dm } G \text{ cr } C \text{ dm } G \text{ cr } D \text{ dm } G \text{ cr } E \text{ dm } G \text{ as } F \text{ ag } A \]

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

![Diagram showing design diagram matching](image)

A in B  B in D  B in E  B in C  G cr E  F ag A

Version 1
Version 2
Version 3
Version 4
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

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

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

<table>
<thead>
<tr>
<th>Systems</th>
<th>Similarities</th>
<th>CamelS</th>
<th>ND</th>
<th>StrS</th>
<th>Combination</th>
</tr>
</thead>
<tbody>
<tr>
<td>JFreeChart</td>
<td>Precision</td>
<td>65.90%</td>
<td>77.27%</td>
<td>72.72%</td>
<td>95.45%</td>
</tr>
<tr>
<td>v0.5.6-v1.0.13</td>
<td>Recall</td>
<td>67.41%</td>
<td>79.06%</td>
<td>74.41%</td>
<td>97.67%</td>
</tr>
<tr>
<td>XercesJ</td>
<td>Precision</td>
<td>84.61%</td>
<td>38.46%</td>
<td>57.69%</td>
<td>92.30%</td>
</tr>
<tr>
<td>v1.0.1-v2.9.0</td>
<td>Recall</td>
<td>88.00%</td>
<td>40.00%</td>
<td>60.00%</td>
<td>96.00%</td>
</tr>
</tbody>
</table>
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?

<table>
<thead>
<tr>
<th></th>
<th>Bug-prone classes</th>
<th>Clean classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{\mu_A}$</td>
<td>973</td>
<td>763</td>
</tr>
<tr>
<td>$S_{\mu_A}$</td>
<td>148</td>
<td>301</td>
</tr>
</tbody>
</table>

Fisher’s test ($p$ – value) $2.2e^{-16}$
Odd-ratio (OR) $2.59$

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

<table>
<thead>
<tr>
<th></th>
<th>Design defect-prone classes</th>
<th>Clean classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{\mu_D}$</td>
<td>1305</td>
<td>431</td>
</tr>
<tr>
<td>$S_{\mu_D}$</td>
<td>210</td>
<td>239</td>
</tr>
</tbody>
</table>

Fisher’s test ($p$ – 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?

<table>
<thead>
<tr>
<th>Systems</th>
<th>PADL</th>
<th>EPI</th>
<th>Step 2</th>
<th>Step 3</th>
<th>Step 4</th>
<th>Step 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArgoUML</td>
<td>7.047</td>
<td>18,098.000</td>
<td>4.835</td>
<td>10.651</td>
<td>10.140</td>
<td>908.329</td>
</tr>
<tr>
<td>DNSjava</td>
<td>2.249</td>
<td>44.209</td>
<td>0.862</td>
<td>0.935</td>
<td>0.075</td>
<td>7.150</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>2.197</td>
<td>62.268</td>
<td>3.135</td>
<td>1.907</td>
<td>0.099</td>
<td>50.030</td>
</tr>
<tr>
<td>Rhino</td>
<td>2.150</td>
<td>50.350</td>
<td>1.783</td>
<td>0.450</td>
<td>0.064</td>
<td>7.985</td>
</tr>
<tr>
<td>XercesJ</td>
<td>4.520</td>
<td>179.410</td>
<td>1.273</td>
<td>0.549</td>
<td>0.032</td>
<td>15.488</td>
</tr>
</tbody>
</table>

| 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”.

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)


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


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

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) ⇒ They could not help developers prioritise their changes according to the forecast scope of changes


**Approach:** Seismology-inspired Metaphor

<table>
<thead>
<tr>
<th>Active seismic areas</th>
<th>“Important” classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Earthquake</td>
<td>Software change</td>
</tr>
<tr>
<td>Epicenter</td>
<td>“Important” changed class</td>
</tr>
<tr>
<td>Seismic wave propagation</td>
<td>Change propagation</td>
</tr>
<tr>
<td>Damaged sites</td>
<td>“Impacted” classes</td>
</tr>
<tr>
<td>Distance from an epicenter</td>
<td>Class level</td>
</tr>
</tbody>
</table>
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

```
<table>
<thead>
<tr>
<th>Time</th>
<th>Commit</th>
<th>Author</th>
<th>Date</th>
<th>Time</th>
<th>Zone</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 min</td>
<td>r1039495</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:24:49</td>
<td>-0500</td>
<td>/xerces/java/trunk/A.java</td>
</tr>
<tr>
<td></td>
<td>r1039496</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:25:00</td>
<td>-0500</td>
<td>/xerces/java/trunk/B.java</td>
</tr>
<tr>
<td></td>
<td>r1039497</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:25:49</td>
<td>-0500</td>
<td>/xerces/java/trunk/C.java</td>
</tr>
<tr>
<td></td>
<td>r1039498</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:26:49</td>
<td>-0500</td>
<td>/xerces/java/trunk/A.java</td>
</tr>
<tr>
<td></td>
<td>r1039499</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:27:04</td>
<td>-0500</td>
<td>/xerces/java/trunk/C.java</td>
</tr>
<tr>
<td></td>
<td>r1039500</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:28:00</td>
<td>-0500</td>
<td>/xerces/java/trunk/B.java</td>
</tr>
<tr>
<td></td>
<td>r1039501</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:33:09</td>
<td>-0500</td>
<td>/xerces/java/trunk/D.java</td>
</tr>
<tr>
<td></td>
<td>r1039502</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:40:49</td>
<td>-0500</td>
<td>/xerces/java/trunk/A.java</td>
</tr>
<tr>
<td></td>
<td>r1039503</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:41:00</td>
<td>-0500</td>
<td>/xerces/java/trunk/C.java</td>
</tr>
<tr>
<td></td>
<td>r1039504</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:42:40</td>
<td>-0500</td>
<td>/xerces/java/trunk/B.java</td>
</tr>
<tr>
<td></td>
<td>r1039505</td>
<td>mrglavas</td>
<td>2010-11-26</td>
<td>13:42:49</td>
<td>-0500</td>
<td>/xerces/java/trunk/A.java</td>
</tr>
</tbody>
</table>
```
Empirical Study Results (1/2)

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

![Diagram 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?

<table>
<thead>
<tr>
<th>Levels</th>
<th>Range 1</th>
<th>Range 2</th>
<th>Range 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6.4015</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10.8485</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>24.8333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>50.2789</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>83.7273</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>895.2652</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Xerces-J: Duncan’s test applied on “number of changes”

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

<table>
<thead>
<tr>
<th>Max Level</th>
<th>Range 1</th>
<th>Range 2</th>
<th>Range 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>10.5333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>16.3333</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>21.6667</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>30.0033</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>43.2000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>54.8667</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Xerces-J: Duncan’s test applied on “number of earthquakes”
Lessons learned ...

- Seismology provides an interesting metaphor for identifying the scope of change propagation.

- The scope of change propagation could reach the 6th 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]


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

```plaintext
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)};

```
**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

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

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

<table>
<thead>
<tr>
<th>Concepts of Immune System</th>
<th>In Biology</th>
<th>In Software</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Body</strong></td>
<td>Software design</td>
<td></td>
</tr>
<tr>
<td><strong>Immune system</strong></td>
<td>Design defects detection approach</td>
<td></td>
</tr>
<tr>
<td><strong>Antigen</strong></td>
<td>Sequence of quality metrics</td>
<td></td>
</tr>
<tr>
<td><strong>Antibody</strong></td>
<td>Known pattern of quality metrics values (Defect Class)</td>
<td></td>
</tr>
<tr>
<td><strong>Affinity</strong></td>
<td>Similarity measure between sets of metrics values</td>
<td></td>
</tr>
</tbody>
</table>

**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=$(a_{g1}, a_{g2}, ..., a_{gk})$ and an Antibody=$(a_{b1}, a_{b2}, ..., a_{bk})$, given by

$$ED(Ag, Ab) = \sqrt{\sum_{i=1}^{k} (a_{gi} - a_{bi})^2}$$
CLONALG Algorithm

1. Generate Initial pool of antibodies
2. Calculate Affinity
3. Select the best n antibodies
4. Clone the selected antibodies
5. Perform Affinity Maturation
6. Calculate Affinity
7. Select the best clones
8. Update Memory Cells
9. Termination Condition? (Yes/No)
10. Output Data for Classifier
Empirical Study

<table>
<thead>
<tr>
<th></th>
<th>Classes</th>
<th>KLOCs</th>
<th>Blobs</th>
<th>FDs</th>
<th>SCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gantt Project</td>
<td>188</td>
<td>31</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>XercesJ</td>
<td>589</td>
<td>240</td>
<td>15</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>777</td>
<td>271</td>
<td>19</td>
<td>19</td>
<td>22</td>
</tr>
</tbody>
</table>

Table: System characteristics
Empirical Study Results (1/3)

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

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

<table>
<thead>
<tr>
<th>Subset</th>
<th>Design Defects</th>
<th>False Positives</th>
<th>Precisions</th>
<th>Recalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subset 1</td>
<td>16</td>
<td>1</td>
<td>94.11%</td>
<td>100%</td>
</tr>
<tr>
<td>Subset 2</td>
<td>16</td>
<td>2</td>
<td>88.23%</td>
<td>100%</td>
</tr>
<tr>
<td>Subset 3</td>
<td>16</td>
<td>2</td>
<td>88.23%</td>
<td>100%</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td>90.19%</td>
<td>100%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>System</th>
<th>Design Defects</th>
<th>False Positives</th>
<th>Precisions</th>
<th>Recalls</th>
</tr>
</thead>
<tbody>
<tr>
<td>GanttProject (on XercesJ)</td>
<td>20</td>
<td>7</td>
<td>65.0%</td>
<td>100%</td>
</tr>
<tr>
<td>XercesJ (on GanttProject)</td>
<td>54</td>
<td>10</td>
<td>81.48%</td>
<td>100%</td>
</tr>
</tbody>
</table>
Empirical Study Results (2/3)

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

<table>
<thead>
<tr>
<th></th>
<th>GanttProject</th>
<th>XercesJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECORE</td>
<td>26.73%</td>
<td>36.22%</td>
</tr>
<tr>
<td>BBN</td>
<td>57.10%</td>
<td>36.50%</td>
</tr>
<tr>
<td>IDS</td>
<td>65.00%</td>
<td>81.48%</td>
</tr>
</tbody>
</table>

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 metaphor
- Studying the type of changes which favors change propagation

Artificial Immune Systems
- Predicting changes which lead to the introduction of bugs
Publications

Journal Papers


Conference Papers


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.