ADvISE: Architectural Decay In Software Evolution

Salima Hassaine, Yann-Gaël Guéhéneuc, Sylvie Hamel, Giuliano Antoniol

SOCCER Lab. and Ptidej Team – DGIGL, École Polytechnique de Montréal, Québec, Canada

March 30, 2012
Context and Motivation

- **Software evolution** is the on-going enhancement of existing software systems, involving both development and maintenance.

- As these systems are adapted to changing requirements, they may suffer from signs of aging.

- Their architecture can deviate substantially from the architecture originally designed i.e., *their architecture tends to decay with time* and it becomes less adaptable to new, emerging requirements.
Architectural Decay

Architectural Decay is defined as “the deviation of actual architecture from planned or original design”. [1,2]


Existing approaches

- **Software Stability**
  - Several authors suggested stability or resilience as a primary criterion for evaluating an architecture.
  - However, none of these approaches have been used to study the signs of software aging or architectural decay.

- **Architectural Decay**
  - Other authors suggested that decayed architectures make their systems more prone to defects and that,
  - in some not-so-rare cases, a system architecture and its implementation code must be thrown away because it is too hard to maintain, unless the decay can be stopped before the architecture is completely unworkable.
Our goal

- Propose a novel approach *ADvISE* and a set of measures to evaluate architectural decay.
- Propose a diagram matching technique to identify structural changes among versions of architectures.
- Propose a set of metrics to identify the signs of architectural decay.
Approach overview
Step 1: Pre-processing

- Reverse-engineer class diagrams from the source code of object oriented programs using **PADL** tool.
- Convert the program model into a string representation using **EPI** tool.

![Diagram](image)

(b) UML-like model

(c) Eulerian model

(d) String representation of the Eulerian model

**Figure:** The conversion of a class diagram into string.
Step 2: Class Renaming Detection (1/3)

Structure-based Similarity

Let $S(C_A)$ and $S(C_B)$ to be the set of methods, attributes, and relationships of class $C_A$ (respectively $C_B$). The structure-based similarity of $C_A$ and $C_B$ is given by:

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

Our algorithm reports, given $C_A$, the $C_B$ with the highest $StrS$ similarity score as the class renamed from $C_A$. 

Step 2: Class Renaming Detection (2/3)

- **Camel-based Similarity**

Let $T(C_A)$ and $T(C_B)$ to be the set of tokens of $C_A$ (respectively $C_B$) names, using a Camel Case Splitter.

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

- **Levenstein-based Similarity**

We use the normalized edit distance ($ND$), given by:

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

where $\text{LEV}$ computes the Levenshtein
Step 2: Class Renaming Detection (3/3)

- **Combination of Similarities**

1. First, we compare the *structure-based similarities* \( (StrS) \) of a candidate renamed class \( C_A \) to many target classes \( \{ C_{B_1}, ..., C_{B_n} \} \).

2. Second, we select the set of target classes having the **highest** \( StrS \) value. Then, we compute their **textual similarities** \( (ND \text{ and } CamelS) \).

3. Third, if we do not find a target class that has the **lowest** \( ND \) and the **highest** \( CamelS \). Then, for each target class \( \{ C_{B_1}, ..., C_{B_n} \} \), we compare \( ND \) and \( CamelS \) similarities to given thresholds.

4. Finally, if none of the target classes has \( ND \) lower than the 0.40 threshold and \( CamelS \) higher than the 0.50 threshold. Then, we can consider that class \( C_A \) was deleted and not renamed.
Step 3: Architectural Diagram Matching

**Bit-Vector Algorithm**

- We build the characteristic vectors of each token in the string representation of *version 1*.
- We read the string representation of *version 2*, triplets by triplets. For each triplets $T$ in *version 2*, we verify whether it exists in *version 1*, as follows:

1) Building the characteristic vectors

\[
A = \underbrace{10000000000000000000000000000}_{30} \\
\text{in} = \underbrace{0101000100010000000000}_{19} \\
B = \underbrace{0010001000100000000000}_{20}
\]

2) Verifying if the conjunction between $(\rightarrow \rightarrow A), (\rightarrow \text{in}), B$ is not null

\[
(\rightarrow \rightarrow A) = \underbrace{01100000000000000000000000000000}_{29} \\
(\rightarrow \text{in}) = \underbrace{0010100010001000000000}_{18} \\
B = \underbrace{0010001000100000000000}_{20} \\
R = (\rightarrow \rightarrow A) \land (\rightarrow \text{in}) \land B = \underbrace{00100000000000000000000000000000}_{29}
\]
Step 4: Architectural Diagram Clustering

Algorithm: Incremental Clustering Principle

1: \( L \leftarrow \text{EmptyList}\{\text{Clusters}\} \)
2: \( S \leftarrow \text{List}\{\text{Common triplets between two program versions}\} \)
3: for each Triplet \( T \) in \( S \) do
4: \hspace{1em} for each Cluster \( C \) in \( L \) do
5: \hspace{2em} if \( T \) is in relations with the existing triplet \( T^* \) in \( C \) then
6: \hspace{3em} if \( T \) is not added to any cluster then
7: \hspace{4em} ADD \( T \) to \( C \).
8: \hspace{3em} ClusterToBe_merged \( \leftarrow C. \)
9: \hspace{2em} else
10: \hspace{3em} MERGE ClusterToBe_mergered to \( C \).
11: \hspace{1em} end if
12: \hspace{1em} end if
13: \hspace{1em} end for
14: if \( T \) is not added to any cluster then
15: Create a new Cluster \( C^* \).
16: ADD \( T \) to \( C^* \).
17: ADD \( C^* \) to \( L \).
18: end if
19: end for
Step 5: Architectural Decay Evaluation

- We perform a pairwise matching of subsequent program architectures.
- We use the sets of stable triplets as indicators of the architectural stability, as follows:
  - **Stability of the original design**: we compute the number of triplets that has a match in all the versions. These triplets are considered to be part of a tunnel, the backbone part of the system.
  - **Stability of the architecture with an enriched functionality**: we compute the number of triplets that have not changed since their first appearance in a given version of a system.
Empirical Study Design

- **Goal**: to show the applicability and usefulness of our approach
- **Purpose**: to provide an approach for identifying class renaming and evaluating architectural decay
- **Quality focus**: the accuracy of the architectural decay evaluation
- **Perspective**: both researchers, who want to study class renaming, and practitioners who analyse software evolution to estimate the effort required for future maintenance tasks.
- **Context**: three open source systems: JFreeChart, Rhino and Xerces-J.
Research Questions (1/2)

**RQ1:** What are signs of architectural decay and how can they be tracked down?

- We investigate whether the numbers of stable triplets are good indicators to measure the architectural decay, and if they provide useful insights to developers regarding the signs of software aging.
Research Questions (2/2)

**RQ2:** Do stable and unstable micro-architectures have the same risk to be fault prone?

This question leads to the following null hypothesis:

- $H_0$: The proportions of faults carried by stable and unstable micro-architectures are the same.
RQ1:

- We perform a pair by pair matching of subsequent program architectures to identify stable triplets in JFreeChart and Xerces-J.
- We study the graph of architectures evolution for each system to assess whether, these indicators provide us useful insights regarding the signs of software aging.
Analysis Methods (2/2)

RQ2:

- To attempt rejecting $H_0$, we test whether the proportion of classes that compose unstable (respectively stable) micro-architectures take part (or not) in significantly more faults than those in stable (respectively unstable) micro-architectures.

- We use the contingency tables to assess the direction of the difference, if any.

- We use Fisher’s exact test, to check whether the difference is significative.
**RQ1:** What are signs of architectural decay and how can they be tracked down?

(a) The evolution of JFreeChart architecture
**RQ1:** What are signs of architectural decay and how can they be tracked down?

(b) The evolution of XercesJ architecture
Study Results (3/3)

**RQ2:** Do stable and unstable micro-architectures have the same risk to be fault prone?

<table>
<thead>
<tr>
<th></th>
<th>Faulty classes</th>
<th>Clean classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unstable classes</td>
<td>105</td>
<td>14</td>
</tr>
<tr>
<td>Stable classes</td>
<td>39</td>
<td>17</td>
</tr>
<tr>
<td>Fisher’s test</td>
<td>0.005</td>
<td></td>
</tr>
<tr>
<td>Odd-ratio</td>
<td>3.244</td>
<td></td>
</tr>
</tbody>
</table>

Table: Contingency table and Fisher test results for unstable classes with at least one fault.

- We can answer to **RQ2** as follows: we showed that stable micro-architectures, belonging to the original design, are significantly less bug-prone than unstable micro-architectures.
Conclusion (1/2)

- Architectural decay is defined as the deviation from an original design, *i.e.*, the violation of architecture caused by the process of evolution.
- When evolution occurs in an uncontrolled manner, the systems become more complex over time and thus, harder to maintain.
- Thus, decayed architectures make their systems more prone to defects We proposed an approach to evaluate the architectural decay.
Conclusion (2/2)

- We proposed a set of structure-based and text-based similarities to identify class renamings in evolving architectures.
- We proposed a bit-vector and incremental clustering algorithms to perform the matching between several versions of an architecture and find stable micro-architectures, which exist in all versions.
- We proposed the numbers of common triplets between several versions of an architecture as indicators of decay, and thus, predictors of fault proneness.
- We also perform a quantitative and two qualitative studies, to show the applicability and usefulness of our approach.
- Future work: Apply our approach to other programs to confirm our observations.
Questions?

Thanks for your attention