Software Architecture Recovery, Smell Detection and Visualization

Computer Science Department
University of Southern California
Los Angeles, CA 90089, USA

Monday, July 10, 2017
Table of Contents

1 Introduction ................................................................................................................................. 3
  1.1 Overview ............................................................................................................................... 3
  1.2 ARCADE Use Case ............................................................................................................... 4
2 Installing ARCADE ..................................................................................................................... 6
  2.1 Requirements ....................................................................................................................... 6
  2.2 Introduction .......................................................................................................................... 6
  2.3 Overview of ARCADE (Source Code and Architecture Diagram) ........................................ 9
3 Architectural Recovery and Smell Detection ............................................................................... 11
  3.1 Conventions used in this and the Following Sections ............................................................ 11
  3.2 ACDC .................................................................................................................................. 12
    3.2.1 Introduction ................................................................................................................... 12
    3.2.2 ACDC Command Reference ......................................................................................... 13
  3.3 ARC .................................................................................................................................... 16
    3.3.1 ARC Workflow overview ............................................................................................. 16
    3.3.2 First Step: Generating a Topic Model ............................................................................ 16
    3.3.3 Second Step: Performing the Analysis – BatchClusteringEngine ................................. 19
  3.4 PKG .................................................................................................................................... 21
    3.4.1 Batch Packager Reference ............................................................................................ 22
    3.4.2 PkgsWithSmellDetection.............................................................................................. 24
4 Analyzing Architectural Changes ............................................................................................... 25
  4.1 Architecture-to-architecture (a2a) ....................................................................................... 25
    4.1.1 Description .................................................................................................................... 25
    4.1.2 Pairwise compute a2a among a set of architectures ...................................................... 26
    4.1.3 Setting the Versioning Scheme ...................................................................................... 27
    4.1.4 Understanding the Output ............................................................................................ 27
  4.2 Cluster coverage (cvg) ........................................................................................................... 28
    4.2.1 Description .................................................................................................................... 28
    4.2.2 Pairwise compute cvg among a set of architectures ..................................................... 28
  4.3 Decay Metrics ....................................................................................................................... 30
    4.3.1 Understanding the output .............................................................................................. 30
5 Analyzing Architectural Smells .................................................................................................. 32
  5.1 Interface and Coupling-based Smells Detection ................................................................. 32
1 Introduction

1.1 Overview

Architecture Recovery, Change, And Decay Evaluator (ARCADE) is a software workbench that employs (1) a suite of architecture-recovery techniques, (2) a catalogue of architectural smell definitions, (3) accompanying smell-detection algorithms, and (4) a set of metrics for measuring different aspects of architectural change and decay. ARCADE combines these elements in the manner depicted in Figure 1 to investigate a variety of questions regarding architectural change and decay.

ARCADE’s foundational element is architecture recovery, depicted as Recovery Techniques in Figure 1. The architectures produced by the Recovery Techniques are used for studying change and decay. ARCADE currently provides access to eight recovery techniques. This allows an engineer to (1) extract multiple architectural views and to (2) ensure maximum accuracy of extracted architectures by highlighting their different aspects.
1.2 ARCADE Use Case

The following section depicts a real-life ARCADE user story to show how ARCADE may be used to bring value to employees assigned to different roles in a software company:

Mr Li uses ARCADE to help Company H

Mr. Li is the Chief Architect at company H, whose development team is planning to upgrade their products’ OS integration from version 6.0 to version 7.0 of mOS, an open-source OS for mobile devices. Since the impact of this upgrade will span a number of product lines, Mr. Li would like to first estimate the extent of architectural changes between the two mOS versions and the architectural quality of the latest version. The goal is that afterwards, Mr. Li and the other technical leaders will be able to make informed decisions regarding the resources needed for this upgrade and where they should concentrate their testing and development efforts.

First, Mr. Li instructs an architect from his team to recover the architectures of both mOS 6.0 and 7.0 using the ACDC and ARC recovery methods, which are integrated into ARCADE. ACDC will give a view of the system architecture based on human-comprehensible patterns commonly encountered in software systems, and those patterns will be used to group code entities into clusters that constitute the architecture of the system. By comparing the makeups of the clusters between the two versions of mOS under the architectural view produced by ACDC, he and his team will get a good sense of how the components and their compositions have changed. This can be a good indicator of where company H's own products need to churn in order to migrate to the new mOS release.

Another recovery method, ARC, employs natural language processing (NLP) to identify recurring topics in the source code, and it uses those topics to group code entities into clusters based on the entities’ distributions of topics. Comparing the results of ARC between the two versions of mOS gives the team another valuable view of how the topics in the architecture have changed in the new mOS release. Specifically, a quick comparison between the architectural views of the two mOS versions produced by ARC indicates that, from version 6.0 to 7.0, the focus has shifted away from system services (e.g., Location service, DB service), as observed in earlier upgrades. This is backed up by the ACDC comparison, which shows that the UI related components have grown in the number of entities between the two versions.

In addition to manual comparisons, ARCADE provides two mechanisms that enable architects to easily identify architectural changes: change metrics and visualization. While the change metrics provide a summarized overview of the changes that is reduced to a set of numerical values, visualization can help the team to study the changes in detail by exploring desired portions of the system.

Next, Mr. Li asks a team member to use ARCADE’s change analysis tools to compute the $a2a$ and $cvg$ metrics between the two versions of mOS. The $a2a$ metric compares two architectures’ overall structures, while $cvg$ focuses on the internal architectures of the system’s components. The resulting $a2a$ values under both the ACDC and ARC views show that the architecture is stable overall (average ~90% similarity between versions 6.0 and 7.0). However, the resulting $cvg$ values (average ~60%
similarity between the two versions) indicate that the changes inside mOS’s components have been extensive.

For a deeper analysis of these results, Mr. Li asks a team member to use ARCADE to visualize the different kinds of changes, such as changes to components’ interfaces and to the component interdependencies. Using ARCADE’s visualization of component structures, the team confirms that most of the changes are related to usability improvements and user experience enhancements. Mr. Li had earlier worried about system services and hardware-level APIs, as these had given his team trouble in the past, but from the visualizations, he can see that these did not change much between the two versions. While Company H does not make major modifications to these impacted areas, the company does create and package several higher-level applications with its products that rely on the user interface design. Therefore, Mr. Li consults with the Project Managers and decides to put more development resources into accommodating the company’s in-house applications to use the new usability and user experience features in mOS 7.0.

ARCADE continues to be useful throughout the development process; for example, it can be used to assess the progress of upgrade and maintenance activities for both engineers and architects. The engineers are empowered by ARCADE’s visualization to periodically re-inspect for new changes to interfaces and dependencies in specific architectural components or implementation entities as mOS 7.0 is being integrated. For example, they found new options to pre-cache web pages, which helps them to improve the user experience of their in-house web browsers.

During the integration, if it is required to modify mOS 7.0’s codebase, Mr. Li and the other architects can use quality metrics implemented by ARCADE to monitor the health of mOS’s architecture. The team decides that the new modifications to mOS should not be allowed to reduce the values of Modularization Quality and Instability metrics by more than 5% of the metrics’ original values. They also point out the vulnerabilities in the architecture (referred to as architectural smells in ARCADE) to engineers. For example, they highlight components that implement too many concerns, and are therefore already or potentially suffering from the Concern Overload smell. This helps engineers to avoid creating new technical debt for the project.

Lastly, the architects use a utility in ARCADE that can find correlations between architectural smells and implementation issues by inspecting the project’s issue tracking system, to identify potential architectural root-causes of those issues. Like other projects in the company, engineers report and manage implementation issues that appear during the integration by using a tool named JIRA. The result of the inspection reveals that many components of the in-house applications are affected by the Duplicated Functionality smell. Mr. Li and the other technical leaders agree that those components should be updated and refactored during the integration.

By using ARCADE, Mr. Li and his team members are able to focus the team’s resources on important areas. ARCADE also helps the team manage the quality of the system’s architecture throughout the development process. Finally, Mr. Li’s project completes two weeks earlier than expected and with a 50% lower defect rate than in the past.
2 Installing ARCADE

2.1 Requirements

The following are the operating systems, execution environments, tools and applications we have used to make sure ARCADE works as described in this manual. Different configurations may work, but have not been tested.

Operating Systems

- Linux Ubuntu 16.04.2 LTS
- macOS Sierra, Version 10.12.5
- Windows 10

Java 8

Note that in the case of Linux Ubuntu, ARCADE was run using the Oracle JVM.

Eclipse Oxygen with PyDev installed

Note that Git can also be used from the command line instead of from within Eclipse.

Follow the latest instructions for installing the PyDev plugin. At the time of this writing, they are to go to Help -> Install New Software -> Add Site -> setting the location to http://pydev.org/updates and then selecting Install -> PyDev for Eclipse.

Python 2 (Optional)

A Python 2 interpreter is necessary to run programs from the ArcadePy subproject. (Note that the programs will not work with Python 3.)

2.2 Introduction

ARCADE is written mainly in Java. ArcadePy, a subproject of ARCADE which provides some additional analyses, is implemented in Python. Unless noted otherwise, any mention of “ARCADE” in this manual is in reference to ARCADE (Java). ARCADE can run as a standalone program using command lines, however it comes packaged with many Eclipse run configurations that serve as examples of the input that needs to be provided to the various tools that are part of ARCADE.

1. The ARCADE project source code can be cloned (downloaded) from the following Bitbucket repository: https://bitbucket.org/joshuaga/arcade
The ArcadePy subproject can be downloaded from:
https://bitbucket.org/joshuaga/arcadepy
In both cases, the “master” branch is the one we use here.

2. To run or develop ARCADE in Eclipse, open Eclipse and select File->Open Projects from Filesystem. Then, press the Directory button and browse to the location of the ‘arcade’ directory; select it and press Ok. The arcade project files and folders should show up as follows:

![Image of Eclipse project view](image)

Figure 2: The ARCADE project viewed in Eclipse

Notes:
- Eclipse can import the ARCADE projects directly using the Git links provided further above: File -> Import... -> Git -> Projects from Git
- For ArcadePy to be imported into Eclipse, PyDev needs to be installed.
- The tools in Arcade can be invoked by command lines. However, for ease of using, we include several run configuration samples in "subject_systems/run_configurations" folder. The following figures show how to invoke the tool via the examples of Eclipse Run Configuration for httpd and Struts2. The first screenshot shows how ACDC can be run on provided versions of Struts2. The second one shows the arguments of that Run Configuration in the Run Configuration editor.
Figure 3: Running Architecture Recovery from Eclipse

Figure 4: Editing Run Configuration Arguments
2.3 Overview of ARCADE (Source Code and Architecture Diagram)

ARCADE is a set of utilities. Two of its most important packages are `edu.usc.softarch.clustering` and `edu.usc.softarch.detection`. One focuses on architecture recovery techniques; the other focuses on smell detection. Class diagrams of the two packages, which are recovered by Enterprise Architect, are as follows:

![Class Diagram of edu.usc.softarch.clustering](image)

Figure 5: Class Diagram of `edu.usc.softarch.clustering`
Figure 6: Class Diagram of edu.usc.softarch.detection
3 Architectural Recovery and Smell Detection

In the research paper, *Toward a Classification Framework for Software Architectural Smells* (D. M. Le, D. Link, Y. Zhao, A. Shahbazian, C. Mattmann, N. Medvidovic), pilot studies of architectural smells in the context of several subject systems suggest that the techniques ACDC, ARC, and PKG are representative of a practical cross-section of all of the various recovery techniques available in ARCADE. The paper further states that these three techniques exhibit good accuracy and scalability, as well as approach recovery from different, complementary angles: ACDC leverages a system’s entity dependencies; ARC relies on retrieved concerns; and PKG reflects the system’s implementation organization. For this reason, we focus on the utilities needed to invoke these three techniques.

In the following sections, we will explain how to recover the architecture of a given software system using the following three recovery techniques:

- ACDC (Arcade/edu.usc.softarch.arcade.AcdcWithSmellDetection)
- ARC (Arcade/edu.usc.softarch.arcade.clustering.BatchClusteringEngine)
- PKG (ArcadePy/BatchPackager.py)

3.1 Conventions used in this and the Following Sections

- This document aims at being self-contained and not requiring the reader to follow any hyperlinks to web URLs after the project has been downloaded and installed. Any additional links are only given for further reference.

- Class, command and script usage references are based on an adaptation of the man page format from ORACLE Solaris (as seen here).

- The command syntax follows the specification for Utility Argument Syntax as found in Section 12.1 of the Open Group Base Specifications Issue 7 (as seen here).

- All shell invocations are assumed to be run from the top-level directory of the arcade project.

- For Java and C/C++ recoveries, two systems with five versions each are used in invocation examples throughout this manual.
  - For Java, it is Apache Struts 2 with versions 2.3.30, 2.3.32, 2.5.2, 2.5.8, and 2.5.10.1 (https://struts.apache.org/)
  - For C/C++, it is Apache HTTPD with versions 2.3.8, 2.4.7, 2.4.10, 2.4.16, 2.4.7 (https://httpd.apache.org/)

- Note that the following files have been provided in the folder hierarchy of the ARCADE project. If all examples in Section 3 are followed in sequence, the results should be exactly as indicated.

<table>
<thead>
<tr>
<th>Item</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Tools</td>
<td>ext-tools</td>
</tr>
<tr>
<td>Example Subject System Source Code for versions of Struts2 and HTTPD</td>
<td>subject_systems/{system name}/src</td>
</tr>
<tr>
<td>Eclipse Run Configurations for recoveries and metrics of the</td>
<td>subject_systems/launch_configurations</td>
</tr>
</tbody>
</table>
After each architectural recovery run, the output files should be separated into different folders for clusters, dependencies and smells. In the subject system examples used in this manual, every attempt has been made to reduce the amount of work the user has to do in preparing the output of a previous command to be used as input for the next; however, separation of files remains a manual step that needs to be performed by the user. (Alternatively, the user can write scripts to perform this task automatically.)

Example:

```
subject_systems/Struts2/output/
  ├── acdc
  │   ├── clusters
  │   ├── deps
  │   └── smells
  │ └── arc
  │   ├── clusters
  │   ├── deps
  │   └── smells
  └── pkg
      ├── clusters
      ├── deps
      └── smells
```

### 3.2 ACDC

**Synopsis:** ACDC uses human-comprehensible patterns commonly encountered in software systems to group source code entities into components.

#### 3.2.1 Introduction

ACDC was introduced by Tzerpos and Holt, who observed that certain patterns for grouping implementation-level entities recur. These patterns include implementation-level entities that

1. reside in the same source file,
2. reside in the same directory,
3. stem from associated body and header files (e.g. .h and .c files in C),
4. are leaves in a system’s graph,
5. are accessed by the majority of subsystems,
6. depend on a large number of other resources, and
7. belong to a subgraph obtained through dominance analysis. This last pattern, called the subgraph dominator pattern, clusters entities that are dominated by a given node together with the dominator node itself.
3.2.2 ACDC Command Reference

edu.usc.softarch.arcade.AcdcWithSmellDetection

Synopsis

AcdcWithSmellDetection inputDirName outputDirName classesDirName
[language]

Description

This class is used to run ACDC on one or more versions of a subject system to recover their architectures and detect architectural smells in the resulting architecture(s).

Parameters

inputDirName

This is a directory containing one or more subdirectories with different versions of the subject system.

For our provided example runs, the first two levels of the hierarchy of the subject_systems/Struts2/src folder look like this:

```
subject_systems/Struts2/src/
├── struts-2.3.30
├── struts-2.3.32
├── struts-2.5.10.1
└── struts-2.5.8
```

outputDirName

This is the directory where all output files go.

Three types of output files are generated:

1. Dependencies RSF(*) files: each line lists one dependency as two fields which represent the source and destination of the dependency. The format of the lines is:

   ```
   depends source destination
   ```

   where `source` and `destination` are code entities.

   Example:

   ```
   depends srclib/apr/file_io/os2/pipe.c srclib/apr/include/apr_lib.h
   ```
2. Cluster RSF files: each line lists an entity-cluster relationship as two fields which represent an entity and the cluster that entity belongs to. The format of the lines is:

   contain  *cluster*  entity

   where  *cluster*  is a cluster and  *entity*  is a code entity.

   Example:

   contain  org.apache.hadoop.dfs.ss  org.apache.hadoop.dfs.DataChecksum

Note that entity-cluster relationships and the corresponding cluster RSF files form the basis of the architectural views described in this manual and are therefore also referred to as “architecture files”.


   (*) “RSF” and the file extension “.rsf” stand for the  Rigi Standard Format,  which is a simple triple format used to describe relationships, such as dependencies, containment and many others.

   NOTE: A set of output files is created for each subdirectory of  inputDirName.  To select a nested subdirectory for analysis, provide its parent directory as  inputDirName.  

   classesDirName

   For Java-based subject systems, this is the directory in each subject system version directory that contains the compiled classes of that system version. This directory needs to exist in all versions.

   If a C/C++ based system is to be recovered, then an empty string should be submitted as an argument for this parameter (see example further below).

   language

   For a C/C++ based recovery, this parameter needs to be set to “c”. For a Java-based recovery, it needs to be omitted.

Example invocations

Shell

# For Struts2 (Java based):
On Linux, macOS and Windows:

```
java -jar edu.usc.softarch.arcade.AcdcWithSmellDetection.jar subject_systems/Struts2/src
subject_systems/Struts2/output/acdc lib_struts
```

Resulting files:

```
subject_systems/Struts2/output/acdc/
├── struts-2.3.30_acdc_clustered.rsf
├── struts-2.3.30_acdc_smells.ser
├── struts-2.3.30_deps.rsf
├── struts-2.3.32_acdc_clustered.rsf
├── struts-2.3.32_acdc_smells.ser
├── struts-2.3.32_deps.rsf
├── struts-2.5.10.1_acdc_clustered.rsf
├── struts-2.5.10.1_acdc_smells.ser
├── struts-2.5.10.1_deps.rsf
├── struts-2.5.2_acdc_clustered.rsf
├── struts-2.5.2_acdc_smells.ser
├── struts-2.5.2_deps.rsf
├── struts-2.5.8_acdc_clustered.rsf
├── struts-2.5.8_acdc_smells.ser
└── struts-2.5.8_deps.rsf
```

# For the Apache HTTP Server (C-based):

On Linux and macOS:

```
java -jar edu.usc.softarch.arcade.AcdcWithSmellDetection.jar subject_systems/httpd/src
subject_systems/httpd/output/acdc "" c
```

On Windows:

```
java -jar edu.usc.softarch.arcade.AcdcWithSmellDetection.jar subject_systems/httpd/src
subject_systems/httpd/output/acdc """" c
```

Resulting files:

```
subject_systems/httpd/output/acdc/
├── httpd-2.3.8_acdc_clustered.rsf
├── httpd-2.3.8_acdc_smells.ser
├── httpd-2.3.8_deps.rsf
├── httpd-2.4.10_acdc_clustered.rsf
├── httpd-2.4.10_acdc_smells.ser
├── httpd-2.4.10_deps.rsf
├── httpd-2.4.16_acdc_clustered.rsf
├── httpd-2.4.16_acdc_smells.ser
├── httpd-2.4.16_deps.rsf
├── httpd-2.4.26_acdc_clustered.rsf
├── httpd-2.4.26_acdc_smells.ser
├── httpd-2.4.26_deps.rsf
├── httpd-2.4.7_acdc_clustered.rsf
├── httpd-2.4.7_acdc_smells.ser
└── httpd-2.4.7_deps.rsf
```
3.3 ARC

Synopsis: ARC uses the words in the source code of a system to detect the concerns the system addresses. It then uses those concerns to group code entities into clusters that represent components.

Architecture Recovery using Concerns (ARC) recovers concerns of implementation-level entities and uses a hierarchical clustering technique to obtain architectural elements. ARC recovers concerns using the statistical language model LDA, which is obtained from the identifiers and comments in a system’s source code. LDA allows ARC to compute similarity measures between concerns and identify which concerns appear in a single implementation-level entity. Thus, ARC attempts to rely on a program’s semantics to perform recovery. ARC represents a software system as a set of documents. A document can have different topics, which are the concerns in ARC. A topic \( z \) is a multinomial probability distribution over words \( w \). A document \( d \) is represented as a multinomial probability distribution over topics \( z \) (called the document-topic distribution). Each implementation-level entity is treated as a document where its document-topic distribution is its feature vector. Hierarchical clustering is performed by computing similarities between entities using the Jensen-Shannon divergence (\( D_{JS} \)), which allows computing similarities between document-topic distributions.

3.3.1 ARC Workflow overview

![Figure 7: The ARC Workflow](image)

3.3.2 First Step: Generating a Topic Model

Our goal is to run ARC on all versions of the application based on a common topic model and inference. For this, we will use the Machine Learning for Language Toolkit (Mallet) from the Computer Science Department at University of Massachusetts - Amherst. Mallet is a Java-based package for statistical natural language processing, document classification, clustering, topic modeling, information extraction and other machine learning applications to text. The Mallet topic modeling toolkit contains efficient, sampling based implementations of Latent Dirichlet Allocation (LDA), Pachinko Allocation, and Hierarchical LDA, which are all useful for analyzing large collections of unlabeled text.

1. The first step is to select a set of versions which gives us a reasonably complete set of almost all possible Topic Models. If the number of versions of the system are relatively small (e.g. less than 10 versions), please select all versions as the set of selected versions and go to the next step; otherwise, please select at least 8 or 9 minor versions of the system to include.
2. Next, copy these versions to a separate directory, `<selectedDir>`.

3. Run the pipeExtractor utility to generate a pipe of the selected versions (see Figure 7). This tool accepts two parameters: `<selectedDir>` `<pipeOutputFolder>`, and it generates file `<output.pipe>` and puts it in the `<pipeOutputFolder>`.

   **Shell Example**

   Note that the respective `subject_systems/{System Name}/output/arc/base` directories are supposed to exist.

   # For Struts2 (Java based):

   On Windows:

   ```
   java -jar .\edu.usc.softarch.arcade.util.ldasupport.PipeExtractor.jar
   .\subject_systems\Struts2\src\.\subject_systems\Struts2\output\arc\base
   ```

   On Linux and macOS:

   ```
   java -jar edu.usc.softarch.arcade.util.ldasupport.PipeExtractor.jar
   subject_systems/Struts2/src subject_systems/Struts2/output/arc/base
   ```

   Resulting files:

   ```
   subject_systems/Struts2/output/arc/base/
   └── output.pipe
   ```

   # For Apache HTTPD server (C-based):

   On Windows:

   ```
   java -jar .\edu.usc.softarch.arcade.util.ldasupport.PipeExtractor.jar
   .\subject_systems\httpd\src\.\subject_systems\httpd\output\arc\base
   ```

   On Linux and macOS:

   ```
   java -jar edu.usc.softarch.arcade.util.ldasupport.PipeExtractor.jar
   subject_systems/httpd/src subject_systems/httpd/output/arc/base
   ```

   Resulting files:

   ```
   subject_systems/httpd/output/arc/base/
   └── output.pipe
   ```

4. Next, we generate Mallet files. The Mallet utility is available in the ext-tools folder of ARCADE (or from the mallet site, [http://mallet.cs.umass.edu/dist/mallet-2.0.7.tar.gz](http://mallet.cs.umass.edu/dist/mallet-2.0.7.tar.gz)). Mallet needs to be run as follows:

   For Linux, macOS and Windows:
Note that in Windows, the environment variable MALLE_HOME needs to be set to the absolute path of the mallet-2.0.7 directory, for example (in the Windows PowerShell):

```
$env:MALLE_HOME = 'C:\Users\Daniel\git\arcade\ext-tools\mallet-2.0.7'`
```

Example for Linux, macOS and Windows:

```
./ext-tools/mallet-2.0.7/bin/mallet import-dir --input subject_systems/httpd/src/ --remove-stopwords TRUE --keep-sequence TRUE --output subject_systems/httpd/output/arc/base/topicmodel.data
```

Resulting files (note that output.pipe came from the previous step):

```
subject_systems/httpd/output/arc/base/
  └── output.pipe
    └── topicmodel.data
```

5. Next, we generate the inference, which contains the trained topic model. Again, we use Mallet but with different options to input the topicmodel.data and generate the infer.mallet file

For Linux, macOS and Windows:

```
./ext-tools/mallet-2.0.7/bin/mallet train-topics --input <TopicModelDirectory>/topicmodel.data --inferencer-filename infer.mallet --num-top-words 50 --num-topics 100 --num-threads 3 --num-iterations 100 --doc-topics-threshold 0.1
```

Example for Linux, macOS and Windows:

```
./ext-tools/mallet-2.0.7/bin/mallet train-topics --input subject_systems/httpd/arc/base/topicmodel.data --inferencer-filename subject_systems/httpd/output/arc/base/infer.mallet --num-top-words 50 --num-topics 100 --num-threads 3 --num-iterations 100 --doc-topics-threshold 0.1
```

*Note*: Mallet 2.0.7, as included with Arcade, has been modified to provide deterministic results. Its use is recommended if reproducible results are desired.

Resulting files:

```
subject_systems/httpd/output/arc/base/
  ├── infer.mallet
  ├── output.pipe
  └── topicmodel.data
```

(Note that output.pipe and topicmodel.data were generated in the two previous steps.)

6. The final step – if needed(∗) – is to copy <output.pipe> and <infer.mallet> into the <project folder>/base directory, where <project folder> equates to outputDirName when running the BatchClusteringEngine below. Be careful to name the subdirectory ‘base’ since this name has been hardcoded to be expected by the tools. Now, we are ready to run the analysis.
* Note that in our examples, the output files are already in the ‘base’ directory.

3.3.3 Second Step: Performing the Analysis – BatchClusteringEngine

edu.usc.softarch.arcade.clustering.BatchClusteringEngine

Synopsis

BatchClusteringEngine inputDirName outputDirName classesDirName [language]

Description

This class is used to run ARC on one or more versions of a subject system to recover their architectures.

Parameters

inputDirName

This is a directory containing one or more subdirectories with different versions of the subject system. (For the needed directory structure, please refer to the ACDC example.)

outputDirName

This is the directory where all output files go.

This directory also needs to contain a subdirectory called “base” that contains the infer.mallet file produced by mallet and the output.pipe file produced by the pipeExtractor as shown above.

classesDirName

If a C/C++ based system is to be recovered, then an empty string should be submitted as an argument for this parameter (see example further below).

For Java-based subject systems, this is the directory in each subject system version directory that contains the compiled classes of that system version. This directory needs to exist in all versions.

language

For a C/C++ based recovery, this parameter needs to be set to “c”. For a Java-based recovery, it needs to be omitted.

Example invocations

Shell
# For Struts2 (Java based):

```
java -jar edu.usc.softarch.arcade.clustering.BatchClusteringEngine.jar
subject_systems/Struts2/src  subject_systems/Struts2/output/arc
lib_struts
```

On Windows:

```
java -jar
.edu.usc.softarch.arcade.clustering.BatchClusteringEngine.jar
.subject_systems\Struts2\src  subject_systems\Struts2\output\arc
lib_struts
```

# For Apache HTTPD server (C-based):

```
java -jar edu.usc.softarch.arcade.clustering.BatchClusteringEngine.jar
subject_systems/httpd/src   subject_systems/httpd/output/arc ""  c
```

On Windows:

```
java -jar
.edu.usc.softarch.arcade.clustering.BatchClusteringEngine.jar
.subject_systems\httpd\src   subject_systems\httpd\arc ""  c
```
Resulting files:

```
subject_systems/httpd/output/arc/
    ├── base
    │     ├── httpd-2.3.8_46_topics_71_arc_clusters.rsf
    │     ├── httpd-2.3.8_arc_smells.ser
    │     ├── httpd-2.3.8_deps.rsf
    │     ├── httpd-2.4.10_46_topics_71_arc_clusters.rsf
    │     ├── httpd-2.4.10_arc_smells.ser
    │     └── httpd-2.4.10_deps.rsf
    ├── httpd-2.4.16_46_topics_71_arc_clusters.rsf
    │     ├── httpd-2.4.16_arc_smells.ser
    │     └── httpd-2.4.16_deps.rsf
    └── httpd-2.4.26_50_topics_82_arc_clusters.rsf
```

(Note that the “base” subdirectory is left over from the previous steps.)

### 3.4 PKG

**Synopsis:** PKG is a straightforward clustering technique based on the package structure of a software system.

The package structure is considered a reliable view of a system's "implementation architecture". While not indicative of the actual architecture underlying the system, the package structure provides a useful baseline (a "sanity check") for many studies. In PKG, entities which have the same parent package are clustered in the same component.
3.4.1 Batch Packager Reference

Implements PKG

Synopsis

batchpackager.py [-h] --startdir start_directory --pkgprefixes package_prefix {package_prefix ...

Description

This tool from ArcadePy is used to obtain the PKG architectural view. BatchPackager requires RSF files that contain dependencies, which are part of the output of ACDC and ARC. This means that ACDC or ARC has to be run before running BatchPackager. For Example:

batchpackager.py --startdir subject_systems/Struts2/acdc/dep --pkgprefixes org.apache.struts2

Parameters

start_directory

This is a directory containing the dependency RSF files to be analyzed. The output will also go here. It includes rsf files containing the clusters computed from packages. Generated architectures have a "_pkgs" suffix.

package_prefix

For Java-based recoveries, the names given are the prefixes of the package(s) of the subject systems to be analyzed (e.g., org.apache.httpd). For C/C++ based recoveries, this should be set to the empty string ("").

Note: Generated Pkg architectures will be in the same folder as dependency files. Those architectures have a “pkgs” suffix. Those files should be moved to a separate folder after recovery.

Examples:

HTTPD / macOS:

python batchpackager.py --startdir ../../../arcade/subject_systems/httpd/output/arc/deps/ --pkgprefixes ""

Resulting files:

subject_systems/httpd/output/arc/deps/
  ├── httpd-2.3.8_deps.rsf
  │    └── httpd-2.3.8_deps_pkgs.rsf
  ├── httpd-2.4.10_deps.rsf
  │    └── httpd-2.4.10_deps_pkgs.rsf
  ├── httpd-2.4.16_deps.rsf
  │    └── httpd-2.4.16_deps_pkgs.rsf
  ├── httpd-2.4.26_deps.rsf
  │    └── httpd-2.4.26_deps_pkgs.rsf
  └── httpd-2.4.7_deps.rsf
    └── httpd-2.4.7_deps_pkgs.rsf
(Note that the ..._deps_pkgs.rsf files are the output of the PKG recovery method.)
3.4.2 PkgsWithSmellDetection

edu.usc.softarch.arcade.PkgsWithSmellDetection

- Detect smells from the output of any recovery method.

Note: While the name of this class may suggest that this is running the PKG recovery method and then detects smells. In reality, it can take the output of any of the three recovery methods ACDC, ARC and PKG as its input.

Synopsis

PkgsWithSmellDetection [-c|--clustersDir clusters_directory] 
-d|--depsDir dependencies_directory -s|--smellsDir smells_directory]|--help

Description

This class takes a directory of clustered rsf files (e.g., cluster files produced by the batch packager) and a directory of dependencies rsf files (these dependencies must coincide with the clustered rsf files) to produce a directory containing corresponding smell files. In fact, any clusterings stored as RSF files (not just for the PKG view) will do for this tool.

Parameters

clusters_directory
directory with cluster RSF files

dependencies_directory
directory with dependency RSF files

smells_directory
Output directory for smell (.ser) files

--help
Prints usage information to the screen.
4 Analyzing Architectural Changes

Once architecture recovery is completed for a software system, the comparison between the recovered architectures of its different versions yields further valuable information about where to look in the system for identifying architectural decay, or even for planning integration and testing efforts. This section reviews some of the tools in ARCADE which gather change metrics for analyzing the differences between system versions.

Architectural change can be considered at two different levels: system-level and component-level. At the system-level, architectural change refers to the addition, removal, and modification of components; at the component-level, architectural change reflects the placement of a system’s implementation-level entities inside the architectural components (i.e., clusters). Studying architectural change at these two levels of abstraction allows us to determine when a system-level architectural view evolves significantly differently than a component-level view. Identifying such discrepancies may reveal points in a software system’s evolution where architectural maintenance issues occur, as well as the scope of those issues.

Note: Before analyzing, the output of Recovery process should be separated into different folders. Usually, there are three different folders which contain smell files (.ser), architecture files (.rsf) and dependency files (.rsf)

4.1 Architecture-to-architecture (a2a)

4.1.1 Description

Synopsis: a2a is a system-level similarity metric based on the cost of transforming one architecture to another.

The algorithm of a2a first computes the minimum transforming cost between an architecture of two different versions of the same software system. The formula to compute a2a is shown in Equation 1.

\[
a2a(A_i,A_j) = (1- \frac{mto(A_i,A_j)}{aco(A_i) + aco(A_j)}) \times 100\%
\]

\[
mto(A_i,A_j) = remC(A_i,A_j) + addC(A_i,A_j) + remE(A_i,A_j) + addE(A_i,A_j) + movE(A_i,A_j)
\]

\[
aco(A_i) = addC(A_b,A_i) + addE(A_b,A_i) + movE(A_b,A_i)
\]

In it, \(mto(A_i,A_j)\) is the minimum number of operations needed to transform architecture \(A_i\) into \(A_j\); and \(aco(A_i)\) is the number of operations needed to construct architecture \(A_i\) from a “null” architecture. Functions mto and aco are used to calculate the total number of the following five operations used to transform one architecture into another: additions (addE), removals (remE), and moves (movE) of implementation-level entities from one cluster (i.e., component) to another; as well as additions (addC) and removals (remC) of clusters themselves. Note that each addition and removal of an implementation-level entity requires two operations: an entity is first added to the architecture and only then moved to the appropriate cluster; conversely, an entity is first moved out of its current cluster and only then removed from the architecture.
Details of the algorithm used by a2a are presented in the research paper A Large-Scale Study of Architectural Evolution in Open-Source Software Systems (Behnamghader, Le, Garcia, Link, Shahbazian, Medvidovic).

4.1.2 Pairwise compute a2a among a set of architectures

edu.usc.softarch.arcade.metrics.BatchSystemEvo

- compare architectures over a sequence of system versions using a2a

Synopsis

BatchSystemEvo [-distopt distance_option] | [--help] directory

Description

This class is used to compare architectures over several consecutive versions using a2a. The versioning scheme can be set in FileUtil.sortFileListByVersion in the edu.usc.softarch.arcade.util package. Please refer to section 4.1.3 for details of how to set the versioning scheme.

Parameters

distance_option
Possible values are 1, 2 and 3. The default is 1.

1: Compare all versions with a version distance (vdist) of 1, i.e. each version is compared to its immediate successor.
2: Compare all versions with all other versions.
3: For all possible vdist values from 1 up, start at the first version, compare it with the version at vdist, then compare that version to the version at vdist from it and so on until there are no more versions to compare.

Example comparisons assuming there are five versions v1, v2, v3, v4, v5:

distance_option 1: v1-v2, v2-v3, v3-v4, v4-v5.
distance_option 2: [No need to show]

directory
Directory that SystemEvo will be computed for. Note: this directory should only contain architecture files (_clusters.rsf) and the version number (major.minor.patch) should be included in the file name. For example:

subject_systems/httpd/output/arc/clusters
   ├── httpd-2.3.8_46_topics_71_arc_clusters.rsf
   │   ├── httpd-2.4.10_46_topics_71_arc_clusters.rsf
   │   └── httpd-2.4.16_46_topics_71_arc_clusters.rsf
   └── httpd-2.4.26_50_topics_82_arc_clusters.rsf
httpd-2.4.7_46_topics_70_arc_clusters.rsf

--help
Print usage information.

4.1.3 Setting the Versioning Scheme

The versioning scheme is hard-coded in the Java class edu.usc.softarch.arcade.util.FileUtil. For a different versioning scheme, change either the code in that class or name the directories containing the source code of the different systems according to the default scheme, which conforms to Major.Minor.Patch, or the regular expression pattern:

"[0-9]+\.[0-9]+(\.[0-9]+)*"

4.1.4 Understanding the Output

Example output for Hadoop:

Comparison distance is: 1
SysEvo from release-0.1.0_deps_pkgs.rsf to release-0.1.1_deps_pkgs.rsf: 100.0
SysEvo from release-0.1.1_deps_pkgs.rsf to release-0.2.0_deps_pkgs.rsf: 85.0224152466368
SysEvo from release-0.2.0_deps_pkgs.rsf to release-0.2.1_deps_pkgs.rsf: 100.0
SysEvo from release-0.2.1_deps_pkgs.rsf to release-0.3.0_deps_pkgs.rsf: 94.50381679389312
SysEvo from release-0.3.0_deps_pkgs.rsf to release-0.3.1_deps_pkgs.rsf: 100.0
SysEvo from release-0.3.1_deps_pkgs.rsf to release-0.3.2_deps_pkgs.rsf: 100.0
SysEvo from release-0.3.2_deps_pkgs.rsf to release-0.4.0_deps_pkgs.rsf: 98.96449704142012
SysEvo from release-0.4.0_deps_pkgs.rsf to release-0.5.0_deps_pkgs.rsf: 94.64906184850591
SysEvo from release-0.5.0_deps_pkgs.rsf to release-0.6.0_deps_pkgs.rsf: 95.75680810639645
SysEvo from release-0.6.0_deps_pkgs.rsf to release-0.6.1_deps_pkgs.rsf: 100.0
SysEvo from release-0.6.1_deps_pkgs.rsf to release-0.6.2_deps_pkgs.rsf: 100.0

DescriptiveStatistics:
 n: 80
 min: 29.070387486669034
 max: 100.0
 mean: 96.83073767951913
 std dev: 8.8679539931019
 median: 99.91161192053676
 skewness: -6.12854152932674
 kurtosis: 44.36378803918304

Explanation of output:

SysEvo values can range from 0 to 100, where a higher value means higher similarity.

In the example, each rsf representing a release is compared using the major.minor.patch versioning scheme for 80 versions of Hadoop.
Descriptive statistics for those 80 versions compared using SysEvo are shown at the end of the output. The purpose of the descriptive statistics is to provide information on the distribution's dispersion and range (min, max, mean, median), variance (std dev), the relative shape of the distribution without having to graph it (skewness), and whether there is a problem with outliers in the data set (kurtosis).
4.2 Cluster coverage \((cvg)\)

4.2.1 Description

Synopsis: \(cvg\) is a new change metric we developed to indicate the extent to which two architectures’ clusters overlap. \(cvg\) uses another metric, \(c2c\) (Cluster-to-Cluster), to determine if two clusters from two architectures are similar and can be considered as overlapped clusters.

\(c2c\) is a similarity metric that measures the degree of overlap (in terms of entities) between implementation-level entities contained within two clusters. The formula to compute \(c2c\) is shown in Equation 2.

\[
\text{Equation 2: The } \text{c2c Formula}
\]

\[
c2c(c_i,c_j) = \frac{|\text{entities}(c_i) \cap \text{entities}(c_j)|}{\max(|\text{entities}(c_i)|, |\text{entities}(c_j)|)} \times 100\%
\]

In it, \(\text{entities}(c)\) is the set of entities in cluster \(c\); and \(c_i\) is a cluster from version \(i\) of system \(S\). The denominator is used to normalize the entity overlap in the numerator by the number of entities in the larger of the two clusters. This ensures that \(c2c\) provides the most conservative value of similarity between two clusters.

The formula to compute \(cvg\) is shown in Equation 3.

\[
\text{Equation 3: The } \text{cvg Formula}
\]

\[
cvg(A_1,A_2) = \frac{|\text{simC}(A_1,A_2)|}{|\text{allC}(A_1)|} \times 100\%
\]

\(\text{simC}(A_1,A_2) = \{c_i \mid (c_i \in A_1, \exists c_j \in A_2)(c2c(c_i,c_j) > th_{cvg})\}\)

\(\text{cvg}(A_1,A_2)\) returns the subset of clusters from \(A_1\) that have at least one “similar” cluster in \(A_2\). More specifically, \(\text{simC}(A_1,A_2)\) returns \(A_1\)’s clusters for which the \(c2c\) value is above a threshold \(th_{cvg}\) for one or more clusters from \(A_2\). \(\text{allC}(A_1)\) returns the set of all clusters in \(A_1\).

\(cvg\) allows an engineer to determine the extent to which certain components existed in an earlier version of a system or were added in a later version. For example, consider a system whose version \(v2\) was created after \(v1\), and for which \(\text{cvg}(A1, A2)=70\%\), and \(\text{cvg}(A2, A1)=40\%\). This means that 70% of the components in version \(v1\) still exist in version \(v2\), while 100% − \(\text{cvg}(A2, A1)\) = 60% of the components in version \(v2\) have been newly added.

The details of the algorithm used for calculating \(cvg\) and \(c2c\) are presented in the research paper \textit{A Large-Scale Study of Architectural Evolution in Open-Source Software Systems} (Behnamghader, Le, Garcia, Link, Shahbazian, Medvidovic).

4.2.2 Pairwise compute \(cvg\) among a set of architectures

From the ArcadePy repository package, the simevolanalyzer.py utility will take a directory of clustered rsf files with different versions for your system and compare the versions using the \(c2c\) metric. Refer to the following run configurations to determine how to use simevolanalyzer:

\begin{verbatim}
arcadepy simevolanalyzer.py hadoop acdc release=x.x.x.launch
arcadepy simevolanalyzer.py hadoop arc release=x.x.x.launch
\end{verbatim}

The run configurations write output to logs/root.log. Here is some sample output:

\begin{verbatim}
source coverage of source clusters from 0.1.0 to 0.2.0: 0.75
target coverage of source clusters from 0.1.0 to 0.2.0: 0.461538461538
\end{verbatim}
Given a cluster c1 in version1, the c2c metric is used to determine if there is a cluster c2 in version2 that matches c1 by some threshold. The threshold selected is 66%, meaning c1 needs to match by 66% or higher. The matches are summarized using source and target coverage metrics:

**Source Coverage Metrics:**
- \( \frac{\text{the number of source clusters in version1 matching clusters in version2}}{\text{the number of clusters in version1}} \)

**Target Coverage Metrics:**
- \( \frac{\text{the number of source clusters in version2 matching clusters in version1}}{\text{the number of clusters in version2}} \)

Therefore, the source coverage is saying that 75% of clusters in version 0.1.0 match some cluster in version 0.2.0 by 66% or more. In other words, 75% of clusters in version 0.1.0 are matched by clusters in version 0.2.0. The target coverage says that 46% of clusters in version 0.2.0 match some cluster in version 0.1.0 by 66% or more. In other words, only 46% of clusters in version 0.2.0 match clusters in version 0.1.0.

Intuitively, about 25% of components from 0.1.0 were changed extensively. Additionally, more than half of the components in version 0.2.0 do not resemble components in 0.1.0. Thus, a few components were changed and many components were added between these versions according to ACDC.

**Note:** to calculate c2c average, you can use the Java class edu.usc.softarch.arcade.util.statistic.C2CAverageAnalyze. The tool only takes one input, which is the output of c2c analysis; it will calculate c2c average of all pairs which have distance of 1.

**Example of input:**

G:\Data\Struts_new_acdc\clustered\Minor
('2.0.9', 'struts2-core-2.0.9_acdc_clustered.rsf')
('2.1.0', 'struts2-core-2.1.0_acdc_clustered.rsf')
('2.1.8', 'struts2-core-2.1.8_acdc_clustered.rsf')
('2.2.1', 'struts2-core-2.2.1_acdc_clustered.rsf')
('2.3.1', 'struts2-core-2.3.1_acdc_clustered.rsf')

**Example of output:**

source coverage of source clusters from 2.0.9 to 2.1.0: 0.516129032258
source coverage of source clusters from 2.0.9 to 2.1.8: 0.451612903226
source coverage of source clusters from 2.0.9 to 2.2.1: 0.387096774194
source coverage of source clusters from 2.0.9 to 2.3.1: 0.387096774194

target coverage of source clusters from 2.0.9 to 2.1.0: 0.695652173913
target coverage of source clusters from 2.0.9 to 2.1.8: 0.56
target coverage of source clusters from 2.0.9 to 2.2.1: 0.5

target coverage of source clusters from 2.0.9 to 2.3.1: 0.5

target coverage of source clusters from 2.1.0 to 2.1.8: 0.5

target coverage of source clusters from 2.1.0 to 2.2.1: 0.5

target coverage of source clusters from 2.1.0 to 2.3.1: 0.5

Avg major source to target = 0.7942678
Avg major target to source = 0.8330797
4.3 Decay Metrics

ARCADE provides a powerful foundation for studying a wide variety of architectural phenomena as software systems evolve. By providing metrics for and analyzing the rate of decay found in software architectures as they change over time, we can improve our understanding of the relationship between architectural change and decay on the one hand, and the reported implementation issues faced by architects and developers on the other hand. Our long-term goal is to leverage ARCADE to enable prediction of architectural decay and major architectural change based on available implementation-level information.

The Java top-level Class "edu.usc.softarch.arcade.decay.BatchDecayMetricsAnalyzer" generates summary metrics about the decay of the subject system. Eclipse Run configurations are available that show example inputs for the class. This tool takes two arguments: <folder containing clustered rsf files> <directory containing dependency rsf files that match the clustered rsf files>

Example clusters directory for the 1st argument:

```
~/workspace/arcade/data/hadoop_svn/arc/release-x.x.x/clusters$ ls
hadoop_release-1.0.0_345_topics_306_arc_clusters.rsf
hadoop_release-1.2.0_383_topics_337_arc_clusters.rsf
hadoop_release-1.0.1_361_topics_321_arc_clusters.rsf
hadoop_release-1.2.1_383_topics_337_arc_clusters.rsf
hadoop_release-1.0.2_362_topics_322_arc_clusters.rsf
hadoop_release-1.0.3_362_topics_322_arc_clusters.rsf
```

Example dependencies directory for the 2nd argument:

```
~/workspace/arcade/data/hadoop_svn/deps$ ls
hadoop_release-1.0.0_deps.rsf
hadoop_release-1.0.1_deps.rsf
hadoop_release-1.0.2_deps.rsf
...<output truncated>...
release-0.10.0_deps.rsf
release-0.10.1_deps.rsf
release-0.1.0_deps.rsf
...<output truncated>...
```

e.g., java -jar edu.usc.softarch.arcade.decay.BatchDecayMetricsAnalyzer subject_systems/httpd/acdc/cluster subject_systems/httpd/acdc/dep

NOTE: The analyzer determines which dependency file belongs to a clusters file by matching version numbers in the file names.

4.3.1 Understanding the output

Example output for a system:

```
DEBUG - No. of vertices: 8
DEBUG - No. of edges: 45
DEBUG - # actual edges: 45
DEBUG - # potential edges: 56
INFO - rci: 0.8035714285714286
INFO - no. of two-way pairs: 21
INFO - no. of two-way pairs / all possible pairs: 0.75
INFO - avg stability: 0.4783653846153846
DEBUG - Elapsed time in milliseconds: 194
```
There are five key metrics computed:

1. **RCI (Ratio of Cohesive Interactions)** - Ratio of actual cluster interactions to possible cluster interactions
2. **Two-way dependencies** - Number of pairs of clusters involved in two-way dependencies / number of possible two-way dependencies (aka bi-directional component coupling)
3. **Instability** - Ratio of incoming dependencies of a cluster to the sum of incoming and outgoing dependencies of the cluster
5. **MQ Ratio** - normalized MQ to a value between 0 and 1 - higher is better. MQ ratio is computed by dividing MQ by the number of clusters.

For the example system shown above, RCI is 0.80. The desirable value of RCI is low and the value ranges between 0 and 1. In this case, it is high, and so there are too many dependencies in this system, thus indicating architectural decay.

Similarly, we compute the number of bi-directional component couplings, which is also preferably low and ranges between 0 and 1. Again, the value is high at 0.75, further indicating decay for this system.

Average instability for the system is 0.48, where the value ranges between 0 and 1. We want the instability to be as close to 0 as possible, indicating that the components are not change-prone. Given that the instability value is at mid-range, this metric is neither indicating decay nor the lack of it.

MQ is bounded by the number of clusters. For example, in the above system with 8 clusters, MQ will be no greater than 8. Thus, we normalize by the number of clusters so that we can compare MQ values across versions. The higher the value of MQ, the more modular the system is, and thus, exhibits less signs of decay. Given that normalized MQ is at 0.30 for this system, there seems to be significant signs of decay.
5 Analyzing Architectural Smells

Architectural decay is a commonly occurring phenomenon in long-lived software systems. During a system's lifetime, its architecture gradually degrades via the appearance of architectural "bad smells", which are instances of poorly thought-through design decisions. In the research paper Toward a Classification Framework for Software Architectural Smells (D Le, D Link, Y Zhao, A Shahbazian, C Mattmann, N Medvidovic), architectural smells fall into one of four categories:

A. **Concern-based smells** are caused by inappropriate or inadequate separation of concerns
B. **Dependency-based smells** arise due to the system components' interconnections and interactions
C. **Interface-based smells** are yielded by the deficiency in defining the system components' interconnections and interactions
D. **Coupling-based smells** appear because of logical couplings across the system components

Note: In order to do smell analysis, smell detection should be done first. In ARCADE, dependency and concern-based smells have already been detected automatically while recovering the architectures. Interface and coupling-based smells require additional steps, which will be explained as follows in Section 5.1.

5.1 Interface and Coupling-based Smells Detection

5.1.1 Collect raw data

1. Collect logical dependencies

   For the first step in collecting logical dependencies, we rely on the command line tool, Code Maat, which is used to mine and analyze data from version-control systems (VCS). Code Maat was developed by Adam Tornhill to accompany the discussions in his book, Your Code as a Crime Scene. Since then, his analyses have evolved into https://codescene.io, which automates all the analyses found in Code Maat. From Adam Tornhill, the ideas behind Code Maat are:

   "To understand large-scale software systems we need to look at their evolution. The history of our system provides us with data we cannot derive from a single snapshot of the source code. Instead VCS data blends technical, social and organizational information along a temporal axis that let us map out our interaction patterns in the code. Analyzing these patterns gives us early warnings on potential design issues and development bottlenecks, as well as suggesting new modularities based on actual interactions with the code. Addressing these issues saves costs, simplifies maintenance and lets us evolve our systems in the direction of how we actually work with the code."

   The procedure for running code-maat is:
   o A compiled version of code-maat is included in the ext-tools folder of Arcade
   o Code-maat can also be compiled by following instructions
     o Code Maat is implemented in Clojure, which uses leiningen to perform builds. Download the leiningen installer using:
       wget https://raw.githubusercontent.com/technomancy/leiningen/stable/bin/lein
       And make 'lein' executable using 'chmod 755 lein'. Next run 'lein' using your standard userid to create the ~/.lein directory containing the leiningen jars. Now move the 'lein' executable to a location in the PATH.
     o Run 'lein uberjar' inside the Code Maat directory to create a standalone jar file containing all of the dependencies for Code Maat:
       code-maat-1.0-SNAPSHOT-standalone.jar
o Clone git projects for the subject software to be analyzed locally. For example,
git clone https://github.com/apache/httpd.git

o Use the following command to download the log file:
git log --all --numstat --date=short --pretty=format:'--%h--%ad--%aN' --no-renames > project.log

o Remove any single quotes ('') from the log file if they exist. Perhaps use:
sed "s/'//g" file.txt > file2.txt

o Run code-maat to get the csv file, using git2 format and coupling mode:
java -jar ../code-maat/target/code-maat-1.0-SNAPSHOT-standalone.jar -l ./project.log -c git2 -a coupling > ./project.csv

In coupling mode, the first column (entity) gives us the name of the module, the second (coupled) gives us the name of a logically coupled module, the third column (degree) gives us the coupling as a percentage (0-100), and finally average-revs gives us the average number of revisions of the two modules. To interpret the data, consider the InfoUtils.java module in the following example output:

<table>
<thead>
<tr>
<th>entity,</th>
<th>coupled,</th>
<th>degree,</th>
<th>average-revs</th>
</tr>
</thead>
<tbody>
<tr>
<td>InfoUtils.java,</td>
<td>Page.java,</td>
<td>78,</td>
<td>44</td>
</tr>
<tr>
<td>InfoUtils.java,</td>
<td>BarChart.java,</td>
<td>62,</td>
<td>45</td>
</tr>
</tbody>
</table>

The coupling tells us that each time it’s modified, there is a 78% risk/chance that we'll have to change our Page.java module too. Since there is probably no reason they should change together, the analysis points to a part of the code worth investigating as a potential target for a future refactoring.

o If the project is a Java-based project, the package name should be converted using the following utility. For C/C++ projects, this step is not needed.
  o Go to package: logical_coupling
  o Run the class: cleanUpCodeMaat.java
    Example: java cleanUpCodeMaat.java subject_systems/git_repo_logs
  o Converted files will have _cleaned suffix
  o Note: CleanUpCodeMaat currently is only used for Java Apache project. For example, it convert
core/src/main/java/org/apache/struts2/util/URLDecoderUtil.java
to 'org.apache.struts2.util.URLDecoderUtil' to match with the output of Architecture recovery.

2. Collect Clone - using Arcade
  o Go to package: edu.usc.softarch.arcade.antipattern.detection.interfacebased
  o Run the class: BatchCloneFinder
    Example: java BatchCloneFinder.java /home/dle/csi/Desktop/Arcade/subject_systems/Struts2/src
    /home/dle/csi/Desktop/Arcade/subject_systems/Struts2/clone
    Note: The input directory has to be full path

3. Collect Dependencies using DependencyFinder
  o Go to Package: edu.usc.softarch.arcade.antipattern.detection.interfacebased
  o Go to Class: BatchDepFinder
    Example: java BatchDepFinder.java /home/dle/csi/Desktop/Arcade/subject_systems/Struts2/src
    /home/dle/csi/Desktop/Arcade/subject_systems/Struts2/depfinder lib_struts
    Note: This tool currently only works on Windows. It uses DependencyFinder tool, which is included in 'ext-tools' folder. DEPENDENCYFINDER_HOME environment variable should be set to point to DependencyFinder folder. For example, DEPENDENCYFINDER_HOME = C:\Arcade\ext-tools\DependencyFinder

33
5.1.2 Run Smell Collectors

For detect both interface- and coupling-based smells, run the following Java class

- edu.usc.softarch.arcade.antipattern.detection.interfacebased.
  DependencyFinderProcessing_ExportJSON
- Change the parameters at the beginning of the class or put them in the input. The list of parameters are as follows
  - The main folder of subject systems
  - Path to the folder that contains cluster files
  - Path to the folder that contains the output of DependencyFinder tool
  - Path to the folder that contains the output of Clone collector tool
  - Path to the folder that contains the output of Code-maat tool
  - Package name of the project
  - Path of the output

Example: java DependencyFinderProcessing_ExportJSON.java

```
subject_systems/Struts2/
acdc/cluster
depfinder
clone
struts2_clean.csv
org.apache.struts2
struct2_acdc_interface_smell.csv
```

Two output files are expected: struts2_acdc_interface_smells.csv and struts2_acdc_interfaces_smells.json. The csv file is more human-readable, and the json file is for further automated analysis. Figure 8 shows the screenshot of a csv output. It starts with the summary of each smells in each version. Later, the details of each smells is provided, such as effected components, effected classes, etc.
5.2 Smell-related Analyses

5.2.1 SmellEvolutionAnalyzer

In the current implementation of ARCADE, smell detection has been implemented in both AcdcWithSmellDetection and BatchClusteringEngine. You can then use the SmellEvolutionAnalyzer to see all the smells that were detected.

To analyze the smells that you detected using ACDCWithSmellDetection, BatchClusteringEngine, or steps from section 5.1, you need to run:

```
"edu.usc.softarch.arcade.antipattern.detection.SmellEvolutionAnalyzer". SmellEvolutionAnalyzer takes one argument, which is the directory containing the " .ser" files which contain detected smells. Once you run it, you can see the output in arcade/logs/root.log.
```

Example: java SmellEvolutionAnalyzer java /home/dle/csi/Desktop/Arcade/subject_systems/Struts2/acdc/ser

Output example: bdc stands for Dependency Cycle, buo stands for Link Overload

```
DEBUG - struts-2.3.30_acdc_smells.ser
DEBUG - contains 15 smells
DEBUG - Listing detected smells for filestruts-2.3.30_acdc_smells.ser:
DEBUG - bdc
org.apache.struts2.el.ss,org.apache.struts2.views.freemarker.tags.ss,org.apache.struts
```
5.2.2 Smell Density

Smell density is the ratio of the number of smells detected to the number of clusters in the system. `edu.usc.softarch.arcade.antipattern.detection.SmellDensityAnalyzer` computes that smell density. It takes two arguments:

- 1st argument - A directory containing smells .ser files
- 2nd argument - A directory containing the corresponding clusters rsf files

NOTE: The analyzer determines which smell files belong to a cluster file by matching version numbers in the file names.

Example: java SmellDensityAnalyzer.java

/home/dle/csi/Desktop/Arcade/subject_systems/Struts2/acdc/ser
/home/dle/csi/Desktop/Arcade/subject_systems/Struts2/acdc/cluster

Understand the output: for each version, the smell density and smell coverage is shown.

version: 1.2.14
# smells: 109
# clusters: 340
smell density: 0.3205882352941177
ratio of smelly clusters to total clusters: 0.8970588235294118

version: 1.2.15
# smells: 125
# clusters: 340
smell density: 0.3067647058823529
ratio of smelly clusters to total clusters: 0.9
... <output truncated>...

Smell density stats:
DescriptiveStatistics:
n: 86
min: 0.12021857923497267
max: 0.4533333333333333
mean: 0.3630352903765285
std dev: 0.04490790388261286
median: 0.35976556435336127
skewness: -1.5778990899393053
kurtosis: 9.05380004556865

Clusters ratio stats:
DescriptiveStatistics:
n: 86
min: 0.7964601769911505
max: 0.9508196721311475
mean: 0.868150053310291
std dev: 0.030733646817469186
median: 0.8711126134827121
skewness: -0.2560967763973427
kurtosis: -0.46508212229634041
In each version, the ratios of #smell/#cluster and #smelly_cluster/#cluster are computed. Smelly clusters are clusters that have smells. At the end of the reports, the statistical information of those ratios across all versions are shown. More details can be found here: http://commons.apache.org/proper/commons-math/userguide/stat.html#1.2_Descriptive_statistics

6 Ongoing and future work

6.1 Visualization

6.1.1 Motivation

We posit that software architects and engineers (end-users) can improve their understanding of the architectural changes and decay of the systems on which they are working with the aid of an appropriate visualization. Although some existing efforts are able to visualize recovered architectures, no effort exists that can effectively visualize (1) structural changes in software architectures, (2) changes of components’ dependencies and (3) interfaces, and (4) locations of decay (i.e., detected smells).

6.1.2 Solution

We intend to use different visualization techniques to create a tool that aids engineers in understanding how architectural changes and decay happen along systems’ lifespan. Specifically, we believe that using interactive visualizations and color-coding labels will be significantly helpful for engineers to adopt our research’s approach. To achieve this goal, we use D3.js, a JavaScript library for manipulating documents based on data. D3 allows binding arbitrary data to a Document Object Model (DOM), and then applying data-driven transformations to the document. It also provides many powerful visualization components.

6.1.3 Visualizing Architectural Changes and Decay of Android Framework

In this section, we introduce four different efforts that aim to visualize different aspects of architectural changes and decay. We use an example of Android Framework, which includes the comparison between the architectures of its version 6.0 and 7.0. The current four visualizations are as follows:

1. Visualizing Component Changes:
   Example: http://softarch.usc.edu/~lemduc/d3test/ComponentChart.html

The example represents the architectures of Android Framework, version 7.0 and 6.0. In this example, components are placed around a circle, and dependencies among components are inner lines. To indicate changes, a color coding scheme is used. On the right-hand side (version 6.0), the components, which are removed in version 7.0, are visualized in red. On the left-hand side (version 7.0), the components, which are added since version 6.0, are visualized in green. The yellow components are changed components. The unchanged components are visualized in gray.
2. Visualizing Dependency Changes:
Example: http://softarch.usc.edu/~lemduc/d3test/DependencyChart.html

This visualization has a similar structure to the previous visualization. However, when a user hovers on a component, the dependencies of that component are highlighted. The red lines represent incoming dependencies, the green lines represent outgoing dependencies, and the purple lines represent removed dependencies.

3. Visualizing Interface Changes:
Example: http://softarch.usc.edu/~lemduc/d3test/InterfaceChart.html
This visualization represents interface changes of components. It shows the overview of the architecture. Entities are visualized by small rectangles, and components are rectangles with labels. Entities are color-coded based on how their interfaces change. The blue entities have new interfaces. The red entities are removed old interfaces. The yellow entities both have new interfaces and have removed old ones. Green entities are unchanged entities. By clicking on a concerned component, a user can see the zoomed-in diagram, which provides the detailed changes within a component.

Figure 11 Visualization of Interface Changes

4. Visualizing Smelly Components:
Example: [http://softarch.usc.edu/~lemduc/d3test/SmellChart.html](http://softarch.usc.edu/~lemduc/d3test/SmellChart.html)
This visualization represents smell infected components. By clicking on different check boxes, components, which are affected by selected smells, will be highlighted by red colors. Users can select multiple choices, and components which have at least one selected smell, will be highlighted.
6.2 Issues Analysis and Correlation with Architecture Smells

As presented by the ARCADE recovery tools discussed earlier, recent shifts in research focus regarding architectural recovery have moved to identifying and examining design decisions as a means for determining "why" an architecture looks the way it does, in addition to the "what" an architecture looks like. Using the information obtained from the combined issue and code repositories yields a set of decisions that has been made during the system's evolution. These design decisions can be classified into categories in order to aid in our understanding of how and when they have impacted architectural decay.

A software system's issue repository can be a valuable source of information about its architectural decay. System stakeholders report in issue repositories the bugs, perfective or adaptive changes, discussions about re-engineering the system, etc.

In this section, we present our approach for relating implementation issues with architectural smells based on the data available in Jira. For example, the tracked issues of Hadoop can be found here. And, a similar approach could be applied to other repositories.

The following lists the important information that engineers should track.

1. **Description**: a detailed description of the issue
2. **Type**: when reporting implementation issues, engineers categorize them as different issue types. Some popular issue types are: bug, new feature, improvement (i.e., enhancement to an existing feature), task (needs to be performed).

3. **Status**: each issue in a tracking repository has a status, which indicates where the issue is in its lifecycle. An issue starts as "open", then generally progresses to "resolved" (a resolution has been identified or implemented, and is awaiting verification by the reporter), and finally to "closed" (the work on the issue is complete).

4. **Affects version(s)**: in which the issue has been found

5. **Fix version(s)**: in which the issue was (or will be) fixed

Additionally, a fixed issue should contain the following information

6. **Fixing commits**: the changes applied to the system to resolve the issue.

7. **Fixing duration**: the amount of time from when an issue had been created until it was fixed

6.2.1 **Issue Analysis Workflow**

![Figure 13: Issue Analysis Workflow](image-url)
The issue analysis workflow is shown in Figure 13. We use ARCADE to recover the architecture and to detect the architectural smells of a subject system. In parallel, we also extract the above mentioned important information of issues from Jira. Based on the collected information, issues are mapped to detected architectural smells using the model depicted in Figure 14. First, we find the system versions that the issue affects. Then we find the architectural smells present in those versions. We say the issue is infected by a given architectural smell if and only if (1) both the issue and the smell affect the same version of the system and (2) the resolution of the issue changes files which are involved in the architectural smell. Based on this relationship, we studied if characteristics (e.g. the issue type and the number of fixing commits) of an issue would be correlated with whether the issue is infected by a given architectural smell.

6.2.2 Running Jira Analysis
In order to run this example to see how a Jira Issues repository can be correlated to smells detected in a subject software, you will need to run three different tools from ARCADE:

1. JiraClientPrototype (to extract issues from JIRA)
2. IssuesAnalyzer (to clean up data in collected issues)
3. SmellToIssuesCorrelation (to find out correlations)

Additional details of running these tools will be available in the next release.