DV8 Explorer User Guide 4.0

Table of contents

DV8 installation

DV8 Download

The user can register and sending DV8 request though archdia.com:

- For education version (valid for 3 years, limited to 2000 files): https://archdia.com/pages/download-dv8-academy
    - an academic email is needed to request
- For standard trial versions (valid for 14 days): https://archdia.com/pages/dv8-standard-free-trial
- For free DV8 viewer: https://archdia.com/pages/download-dv8-viewer

After that, the user will receive an email from ArchDia <notifications@fetchapp.com> that contains:
(1) a license key
(2) an Activation code that starts with "acti-".
(3) a link where the installation package can be downloaded. The user can choose a package based on the OS in use

Supported operating systems

DV8 supports the following OS systems and provide corresponding installation packages:

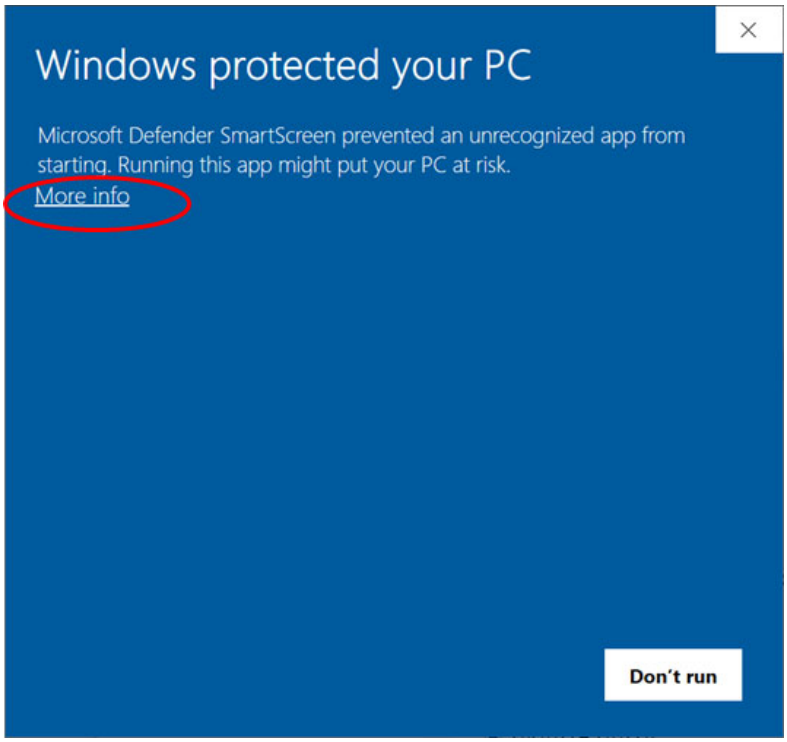
(1) Linux/unix:
DV8_standard_unix_linux-amd64.sh

(2) MacOS 10.13 or above
DV8_standard_macOS.dmg

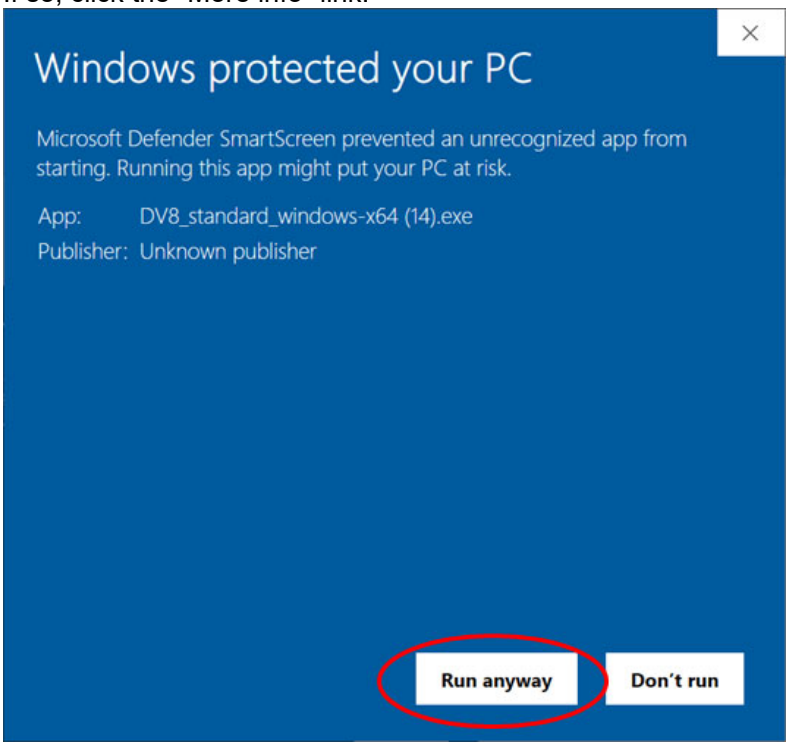
(3) Windows 7 or above
DV8_standard_windows-32.exe
DV8_standard_windows-x64.exe

Installation Process

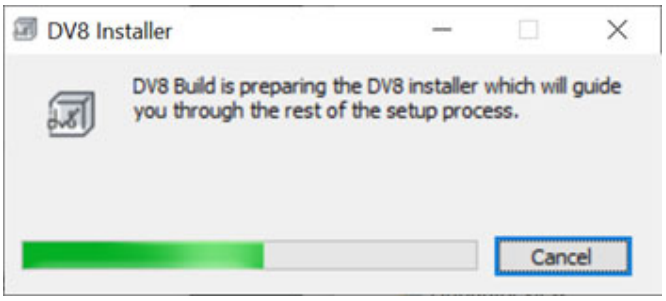
Executing an installation packages will start an standard installer. It is possible to receive a warning as follows:



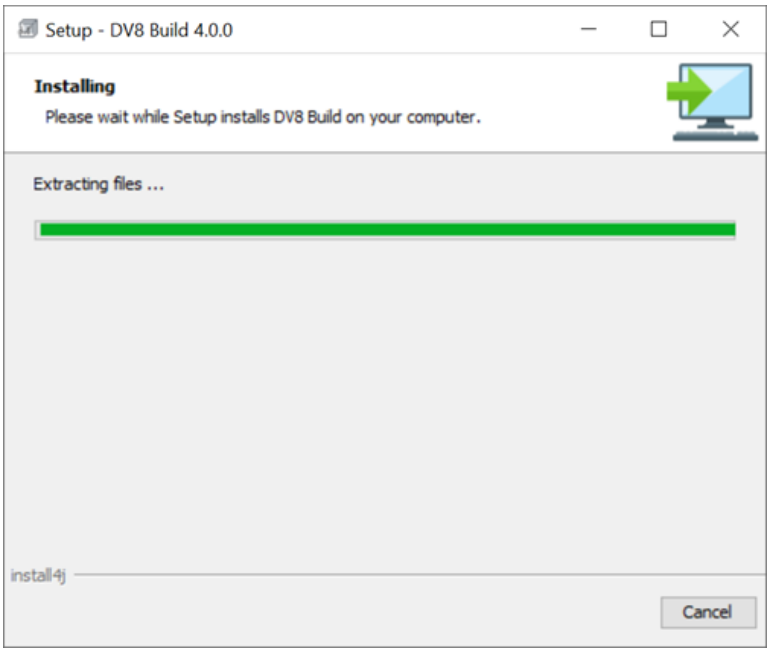
If so, click the "More info" link:

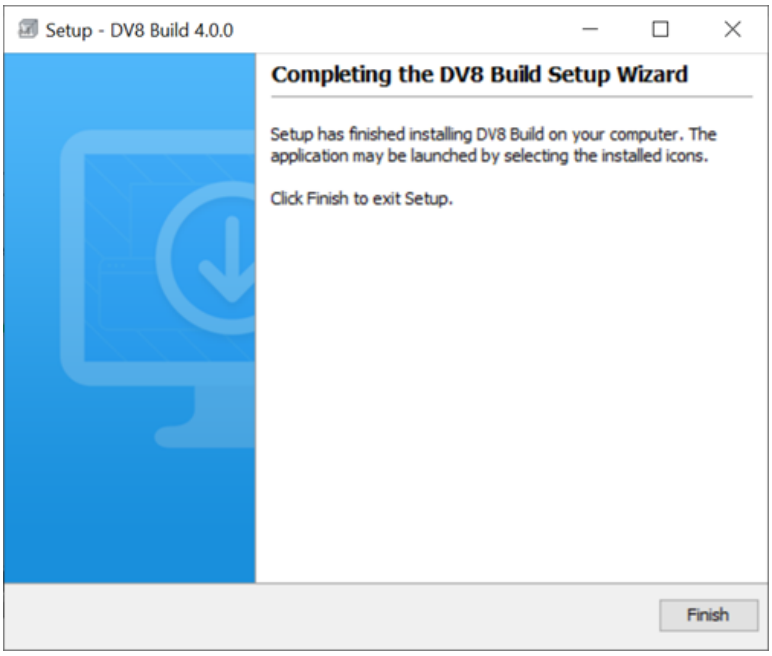


Click the "Run anyway" button, and DV8 installer will start:



The user just needs to follow the prompts to finish the installation process:



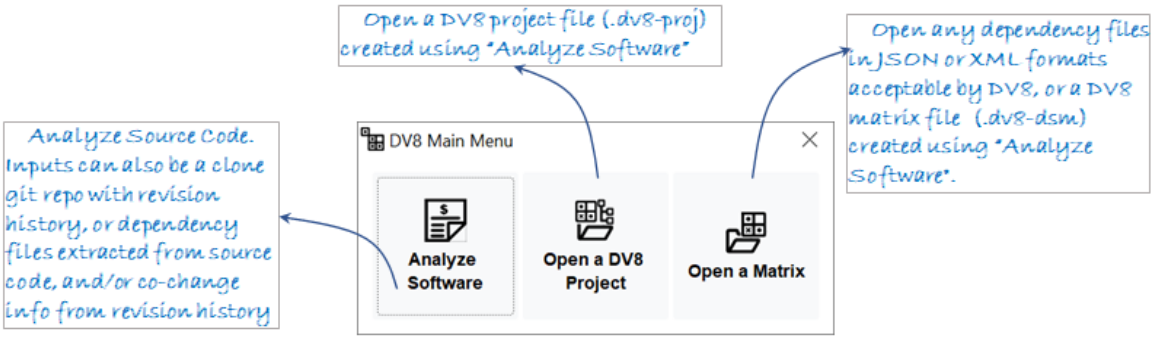


If the installation is successful, the DV8 icons will show up in the Start menu. Clicking the DV8

icon will start the program.

How to analyze software design using DV8

Once you have installed DV8 Explorer, double click the icon, and the following window will show up:



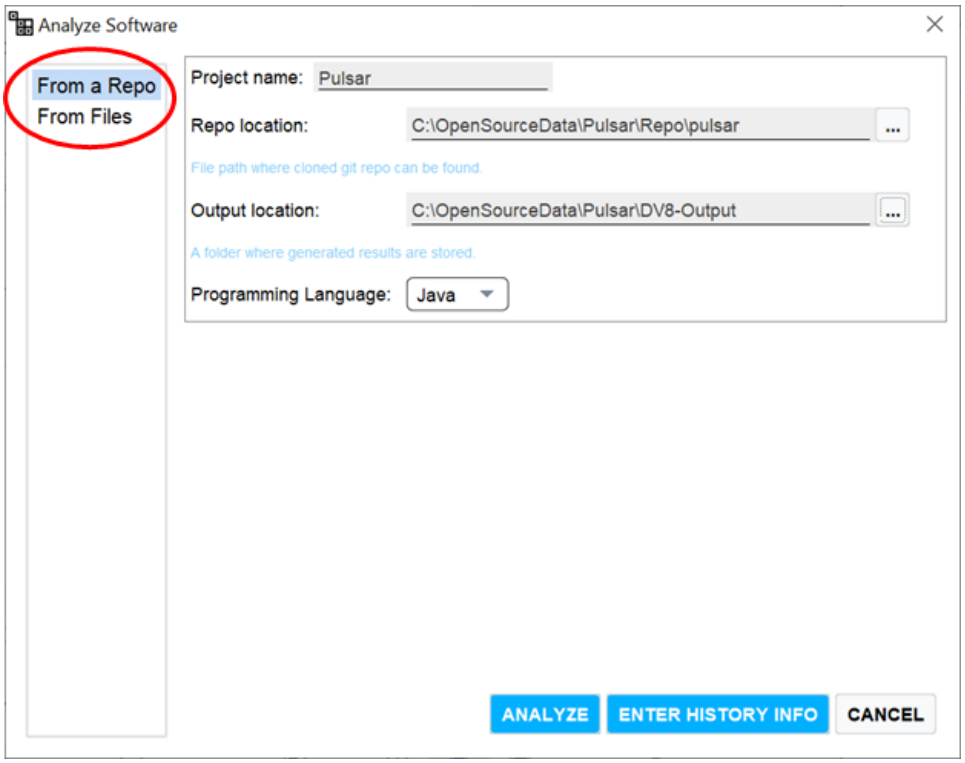
1. Analyze Software
Clicking this button will start a GUI through which the user can analyze source code design automatically, including identifying design anti-patterns, hotspots, and quantify design debt in terms of maintenance costs.

2. Open a DV8 Project
Click this button to open a DV8 project file (.dv8-proj) that was already created by using the “Analyze Software” function.

3. Open a Matrix
Click this button to open any dependency files following *JSON* or *XML* formals acceptable by DV8, or a DV8 matrix file (.dv8-dsm) created by using the “Analyze Software" function.

Analyze Software

Following is the GUI for you to enter input information.



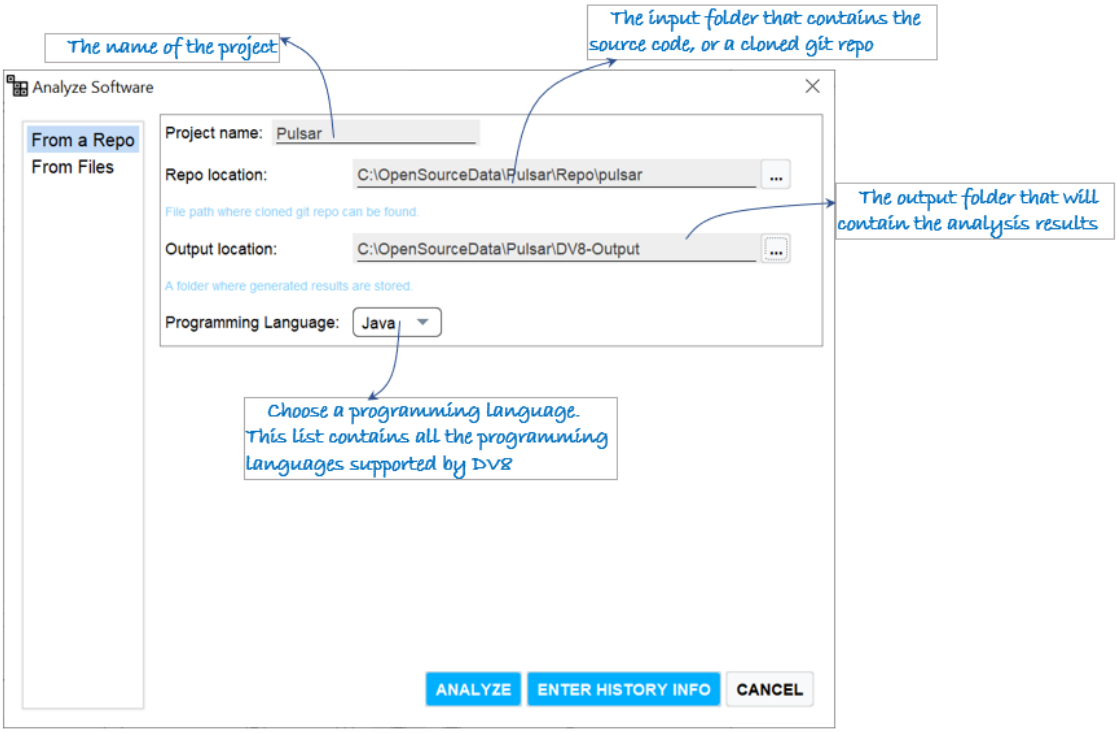
In order to analyze software design, DV8 first needs to (1) extract file dependency information from a folder contain source code or a cloned git repository (use the "From a Repo" tab), or (2) accept existing files containing dependency information (use the "From Files" tab). These files can be extracted using a 3rd party tool.

After that, you can choose to start analyzing by clicking the ANALYZE button, or click the ENTER HISTORY INFO button to the enter evolution history information, either from the git repo, or an exported git log file.

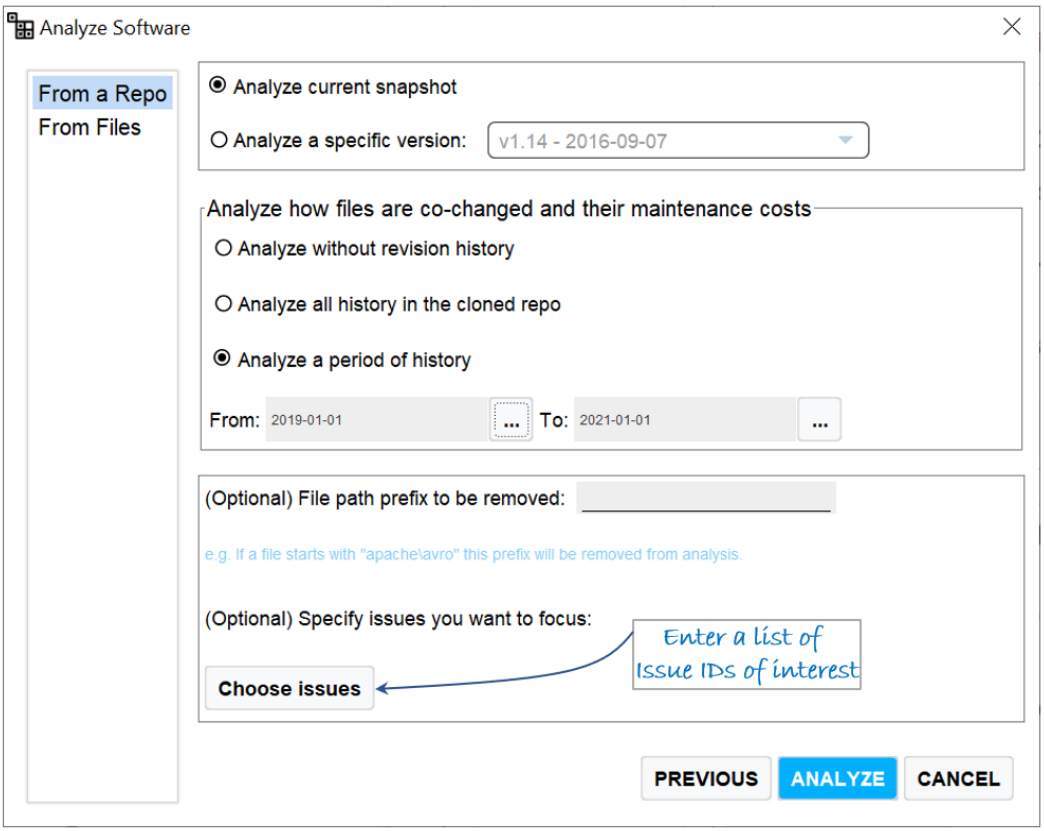
Analyze software from a repo

Using this interface, the user can enter the following information:

- The name of the project
  - DV8 will generate a project file (.dv8-proj) and an architecture analysis report, which will use this name to refer to the project.
- The input folder that contains the source code, or a cloned git repo
- The output folder that will contain the analysis results
- The programming language used in this project.
  - This list includes all the programming languages supported by DV8



After that, you can choose to start analyzing by clicking the ANALYZE button, or click the ENTER HISTORY INFO button to the enter evolution to enter revision history information as shown in the figure below.

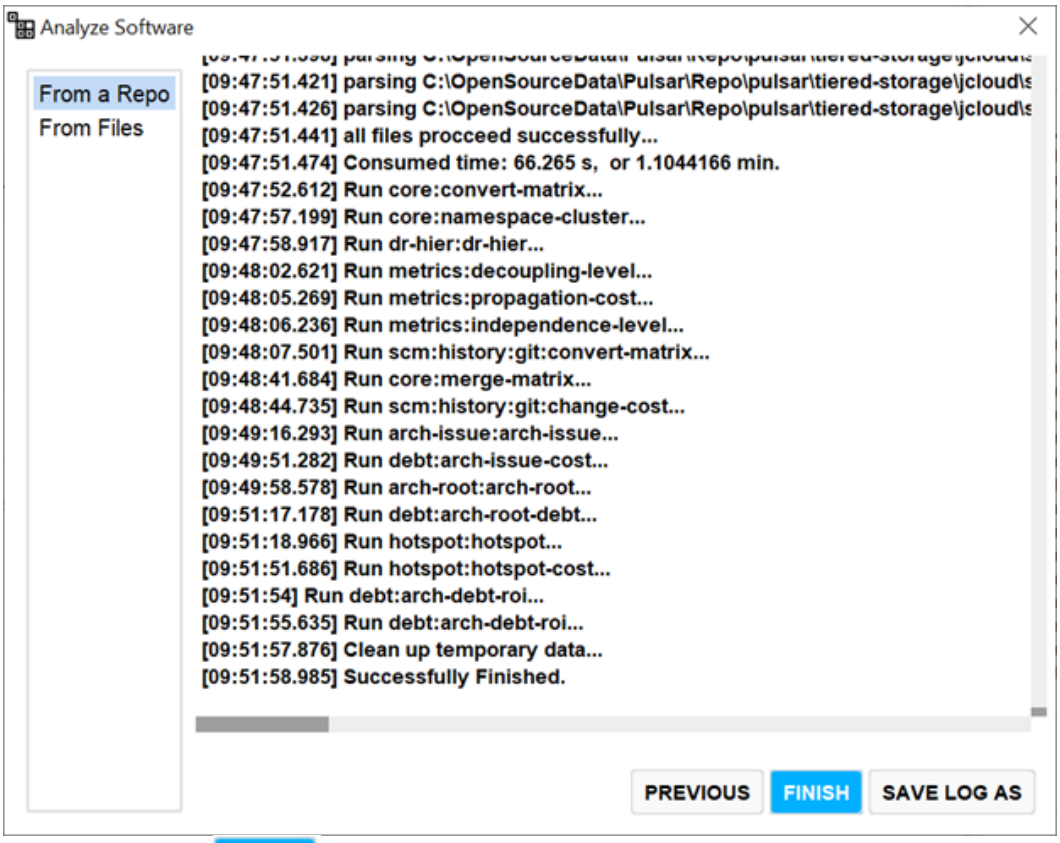


- If the input folder doesn't contain revision history information, or only contains source code, then only the "Analyze without revision history" option is available to click. All other options

are disabled. The user can also choose not to analyze the revision history by choosing this option.

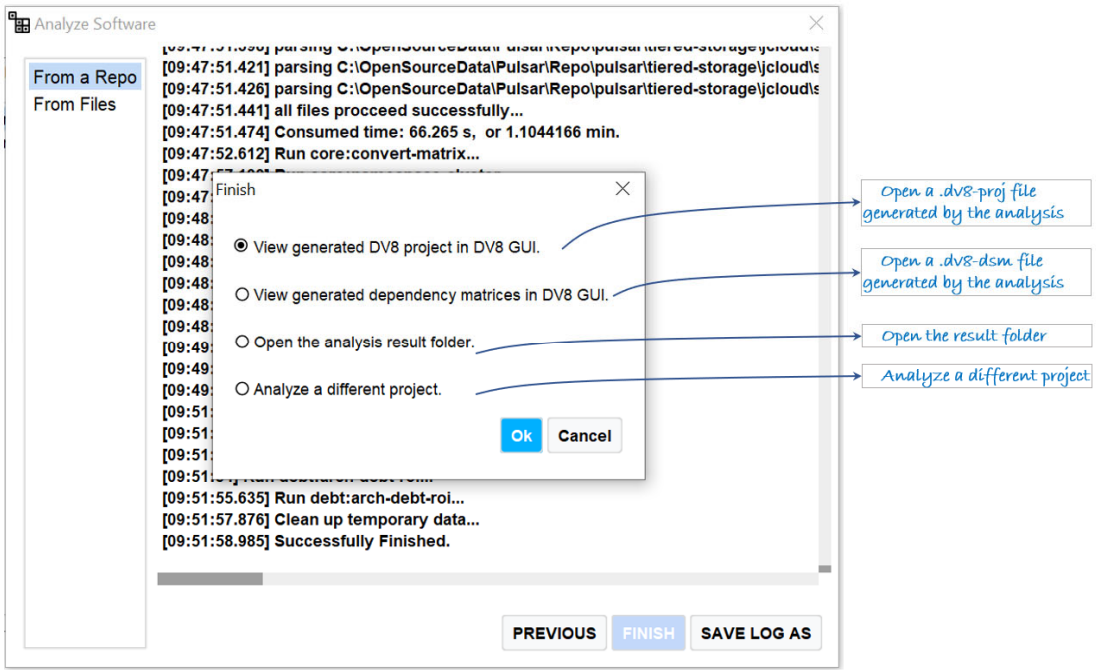
- If the folder is a cloned git repo, then the user can
  - choose to analyze the latest version, or a previously tagged version
  - choose to analyze all revision history, or an evolution period. Please note that for a system with a very long history, choosing a recent evolution period could save time and be more useful.
  - Specific a file path prefix that need to be removed
    - This function is used to match file names in source code and in git repo. For example, a source file in a local folder may start with "\apache\avro\", while in the git repo the file may start with "smith\apache\avro". In this case, the user can enter "\smith\" for this prefix to be removed from the analysis, so that the system can match source files with their names in the repository.
  - In addition, the use can click the **Choose issues** button to load a csv file that contains a list of issue IDs (download an example) as part of the input.
    - If a issue list is entered, DV8 will analyze the design structure related to the given list of issues, for example, calculate maintenance hotspots related to these issues, or the file structures related to a selected set of bugs or features.
    - The user needs to collect these issue IDs manually. If this list is not available, this step can be skipped.
    - In order to identify commits that are linked to issues, the user also need to enter a regular expression indicating an issue ID.

After all the inputs are specified, the user can click the **ANALYZE** button to proceed. If all the inputs are correct, the analyze will finish as follows:



After clicking the FINISH button, the user will have the following options:

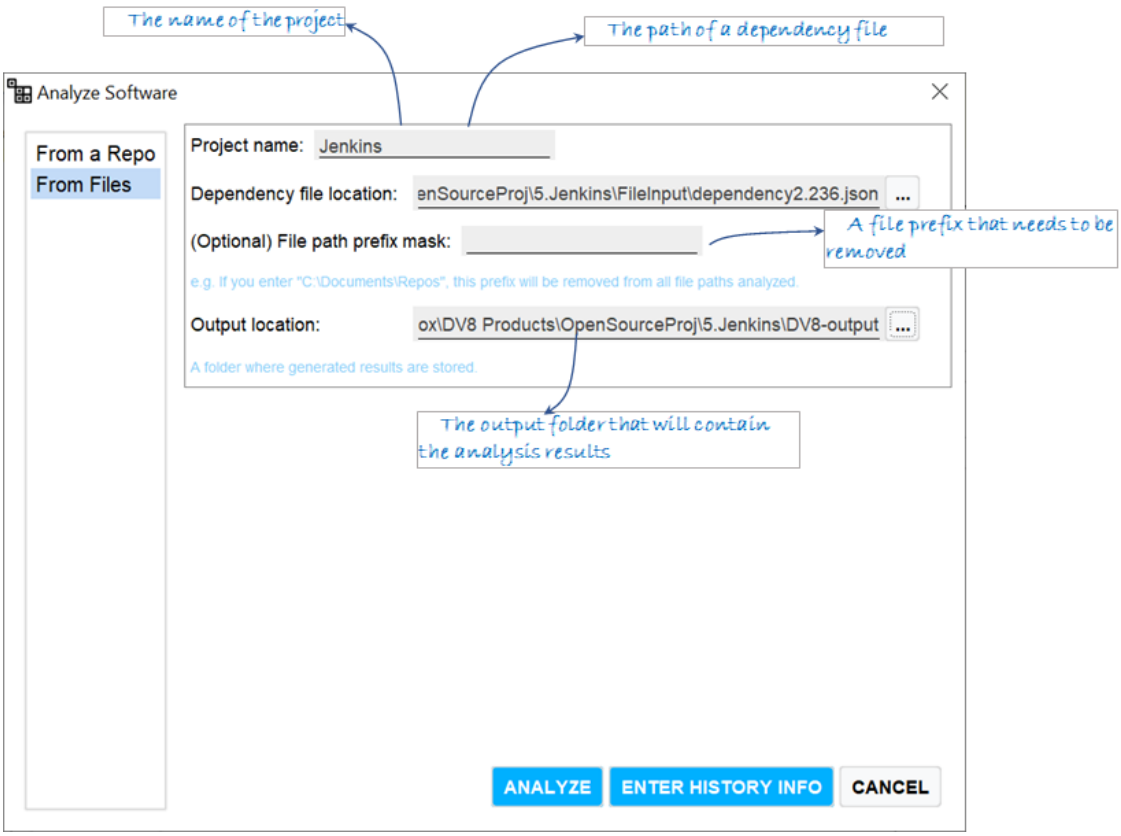
- View generated DV8 project in DV8 Explorer:
  - The analysis will generate a *.dv8-proj* file that can be used to explore dependencies, simulate changes, and link to the detailed source code that can be opened using a DV8 viewer
- View generated dependency matrices in DV8 Explorer:
  - The user can use this option to open a DSM file that only has dependency information but does not contain detailed source code information
- Open the analysis result folder
  - This folder contains all analysis results. Please refer to "Understand the analysis results" section for details.
- Analyze a different project: this option will return to the first GUI allowing the user to analyze another project.



Analyze software from file inputs

Using this interface, the user can analyze dependency file generated by various 3rd party tools. Using this function, the user can analyze dependencies among various software artifacts, such as components, libraries, or test suits, as long as the dependencies among them can be extracted and represented using the standard JSON (download an example) or XML format (download an example)

The user can enter the following information:

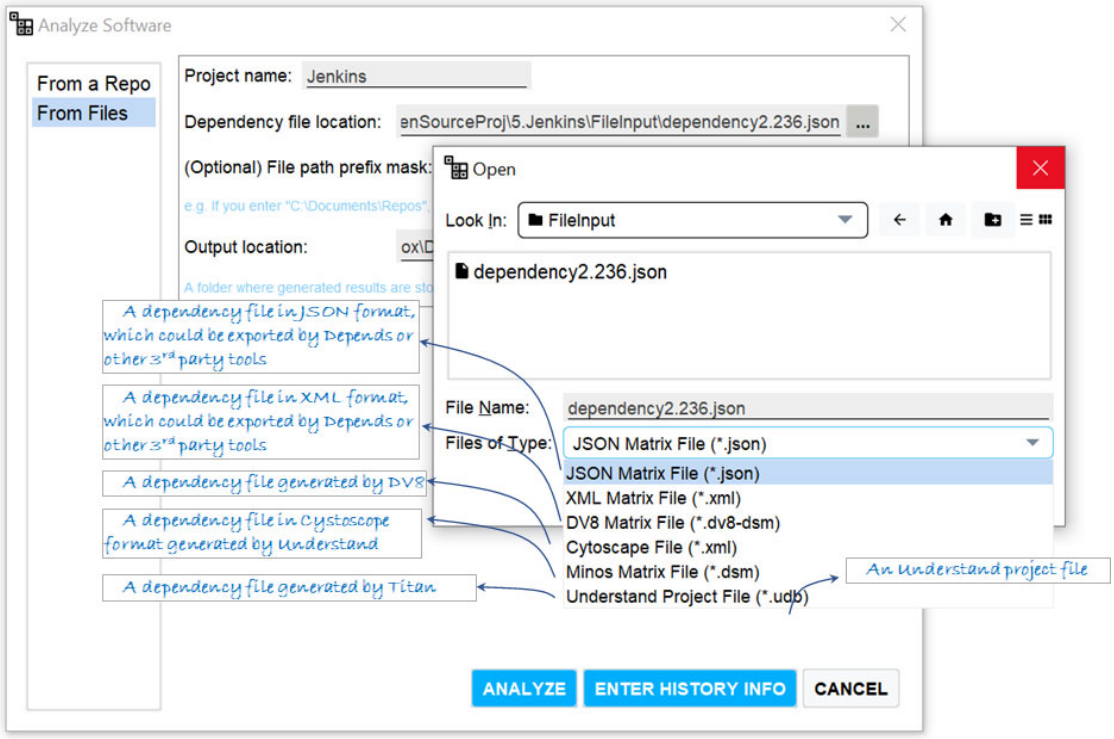


- The name of the project. DV8 will generate a project (*.dv8-proj*) file and an architecture analysis report, which will use this name to refer to the project.

- A dependency file, which can be one of the following formats:
  - A .json file, which could be exported by Depends™ or other 3rd party tools (download an example)
  - A .xml file, which could be exported by Depends™ or other 3rd party tools (download an example)
  - A .dv8-dsm file generated by DV8
  - A Cytoscape .xml file generated by Understand™
    - The user can obtain a Cytoscape dependency file as follows:
      1) Load a project into Understand
      2) Use the following menu to general a Cytoscape XML report

      `Reports -> Dependency -> File Dependencies -> Export Cytoscape XML`
  - A dependency file generated by Titan
  - An Understand™ project file (.udb)

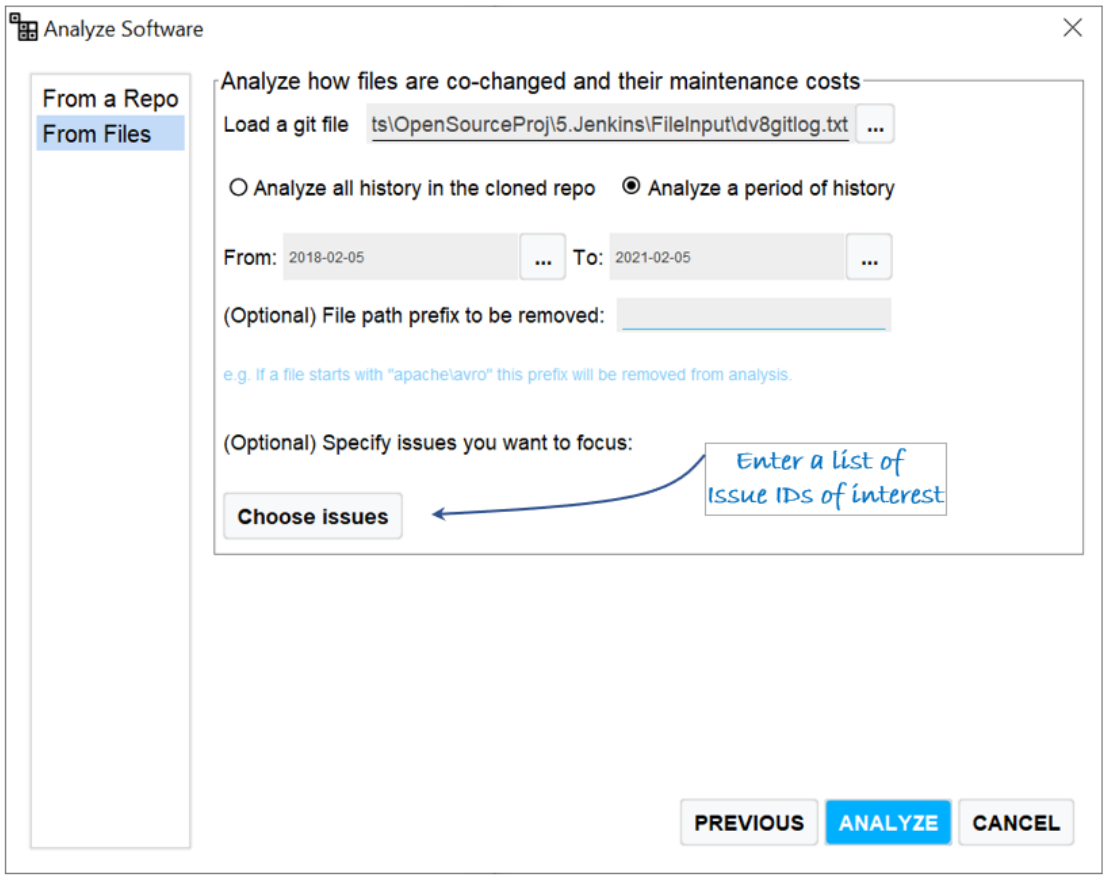
The use can generate dependency information among various artifacts using various tools. As along as it follows the standard JSON or XML format, it can be opened and analyzed using DV8.



- The output folder that will contain the analysis results. Please refer to "Understand the analysis results" section for details.
- The prefix within the file path that needs to be removed.
    - This function is used to match file names in source code and in git repo. For example, a source file in a local folder may start with "c:\opensource\apache\avro\", while in the git repo the file may start with "\apache\avro". In this case, the user can enter "c:\opensource\" for this prefix to be removed from the analysis, so that the system can match source files with their names in the repository.

After the input file is specified, the user can either

- click the ANALYZE button to start analyzing, or
- click the Enter History Info button to enter evolution history information as follows:



- This UI allows the user to load a log file exported from a version control system, which can be generated as follows:
  - the user can use either of the following commands to get records from svn:

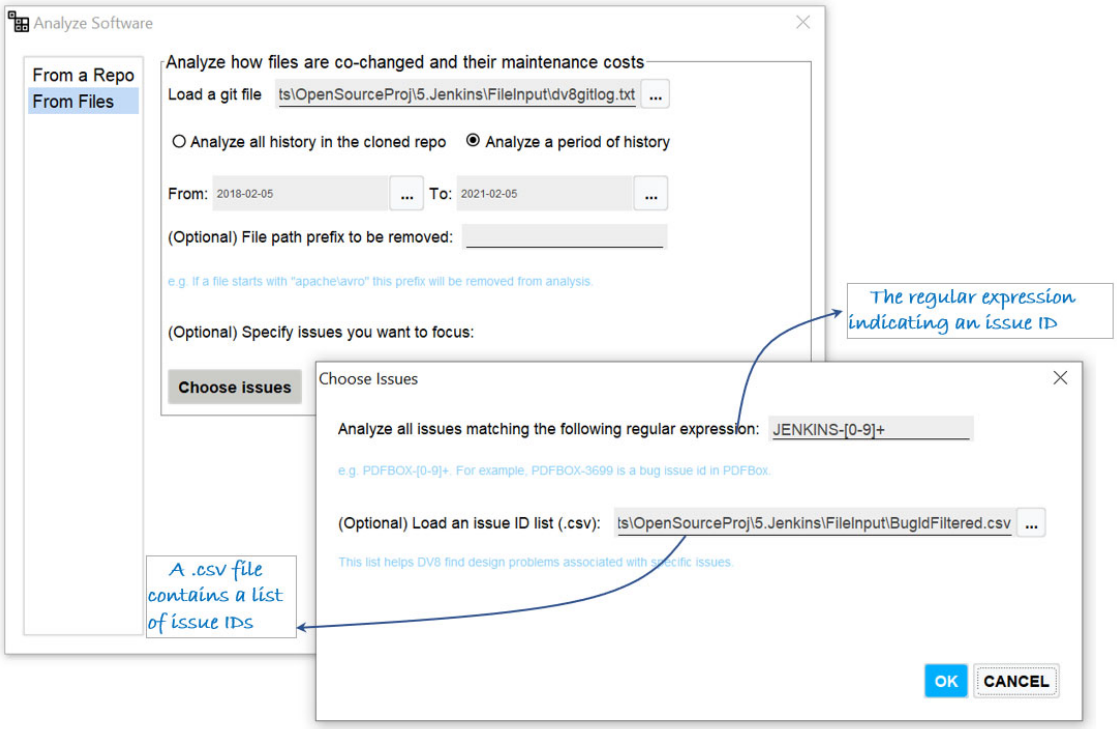
```
svn log --xml -v repo; //this command will return a xml file without churn
information

svn log --diff repo; //this command will return a plain text file with churn
information, that is, how many LOC were added, deleted, or changed.
```

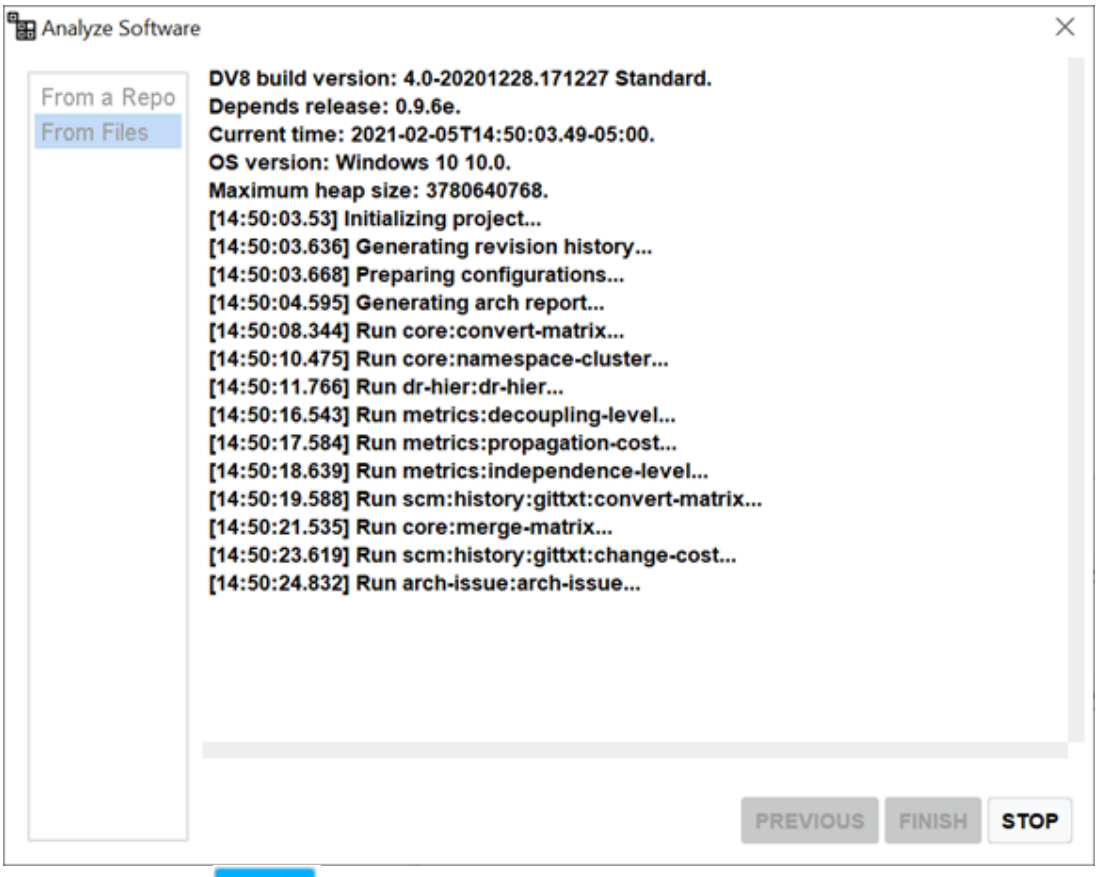
  - Or, the user can use following command has to be used to get records from git:

```
git log --numstat --date=iso //this command will return a plain text file with
churn information, that is, how many LOC were added, deleted, or changed.
```

- The user can also choose analyze all the history in the log file, or a period of it.
- In addition, the use can click the **Choose issues** button to load a csv file that contains a list of issue IDs (download an example) as part of the input.
  - If a issue list is entered, DV8 will analyze the design structure related to the given list of issues, for example, calculate maintenance hotspots related to these issues, or the file structures related to a selected set of bugs or features.
  - The user needs to collect these issue IDs manually. If this list is not available, this step can be skipped.

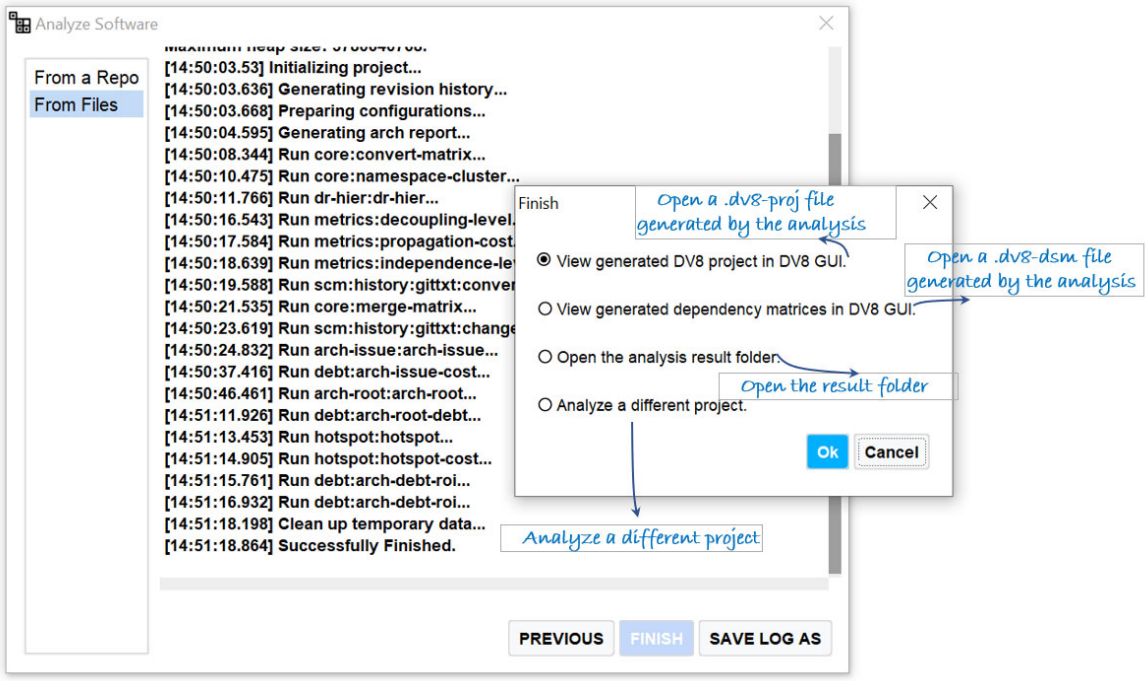


After all the inputs are specified, the user can click the ANALYZE button to proceed. If all the inputs are correct, the analyze will finish as follows:



After clicking the FINISH button, the user will be given the following options:

- View generated DV8 project in DV8 GUI:
  - The analysis will generate a .dv8-proj file that can be use to explore dependencies, simulate changes, and link to the detailed source code info.
- View generated dependency matrices in DV8 GUI:
  - The user can use this option to open a DSM file that only has dependency information but does not contain detailed source code information
- Open the analysis result folder
  - This folder contains all analysis results. Please refer to "Understand the analysis results" section.
- Analyze a different project: this option will return to the first GUI allowing the user to analyze another project.



Understand the analysis results

This folder contains all the analysis results, including the following files and two subfolders:

- A *.dv8-proj* file:
  - This file can be opened using a DV8 GUI. The user can explore dependencies among files, simulate changes, and explore detailed source code information.
- A *depends-output* subfolder:
  - DV8 Explore integrates Depends™ as a dependency extraction tool. This folder contains the following files:
    - *dependency.csv*: this file contains the programming languages and LOC of each source file (download an example)
    - *dependency.json*: this file contains the dependency information among all the source files that can be opened using DV8 GUI (download an example)
    - *depends-dv8map.json*: this file contains all the dependency types among source files that will be used internally by DV8 (download an example)
- A *dv8-analysis-result* subfolder that will be introduced in the next section.

dv8-analysis-result

This folder contains the main DV8 analysis results. These state-of-the-art analysis includes:

- Overall modularity measures, including Decoupling Levels and Propagation Costs
- Design anti-pattern detection
- Hotspot detection
- Root analysis
- Debt quantification and return on investment analysis

Please refer to the Knowledge Base section for their detailed definitions.

The result folder contains the following two files and subfolders:

- The *analysis-summary.html* file:
  - This file summarizes all the analysis results and can be opened using a browser (download an example)
- The *file-measure-report.csv* file:
  - This file contains all the measures for each of the source file (download an example)
- The *anti-pattern* subfolder:
  - Based on recent research, DV8 detects 6 types of design anti-patterns using both structural information and revision history, summarizes each instance of each pattern, and their maintenance costs, into spreadsheets, and generates the dependency matrix of each instance for the user to examine using the DSM viewer. The detailed and summary information are contained in this folder

- The *dsm* subfolder: This folder contains the various auto-generated design structure matrices
- The *hotspot* subfolder: This folder contains the detailed information of hotspots detected by DV8
- The *maintenance-costs* subfolder: This folder contains the following two flles:
  - The *all-file-change-cost.csv* file that lists the CommitID, IssueID, and Churn of each revision of each source file.
  - The *target-issue-id-list.csv* file that lists all the target issue ID, such as a ID list of bug issues or feature issues.
- The *root* subfolder: DV8 automatically calculates a set of correlation matrices covering files that are error-prone or change-prone. The objective is to reveal design issues that lead to high maintenance costs.

Studies have shown that 50% to 90% error-prone files will be concentrated in 5 or fewer file groups.

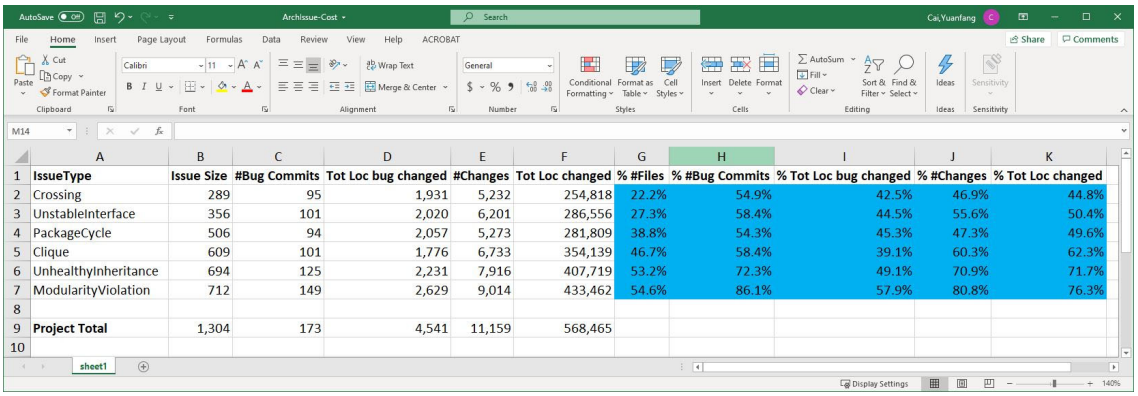
The more error-prone a file is, the more likely that it is connected with other files, resulting in the spread of defects in multiple files. We call these file groups that cover most error-prone files as Root spaces.

anti-pattern

<break time="1s"/>

This folder contains the following two subfolders:

- The *anti-pattern-costs* folder contains the following files:
  - The *anti-pattern-cost.csv*: This spreadsheet summarizes the maintenance cost for each type of anti-pattern.



	A	B	C	D	E	F	G	H	I	J	K
1	**IssueType**	**Issue Size**	**#Bug Commits**	**Tot Loc bug changed**	**#Changes**	**Tot Loc changed**	**% #Files**	**% #Bug Commits**	**% Tot Loc bug changed**	**% #Changes**	**% Tot Loc changed**
2	Crossing	289	95	1,931	5,232	254,818	22.2%	54.9%	42.5%	46.9%	44.8%
3	UnstableInterface	356	101	2,020	6,201	286,556	27.3%	58.4%	44.5%	55.6%	50.4%
4	PackageCycle	506	94	2,057	5,273	281,809	38.8%	54.3%	45.3%	47.3%	49.6%
5	Clique	609	101	1,776	6,733	354,139	46.7%	58.4%	39.1%	60.3%	62.3%
6	UnhealthyInheritance	694	125	2,231	7,916	407,719	53.2%	72.3%	49.1%	70.9%	71.7%
7	ModularityViolation	712	149	2,629	9,014	433,462	54.6%	86.1%	57.9%	80.8%	76.3%
8											
9	**Project Total**	1,304	173	4,541	11,159	568,465					
10											

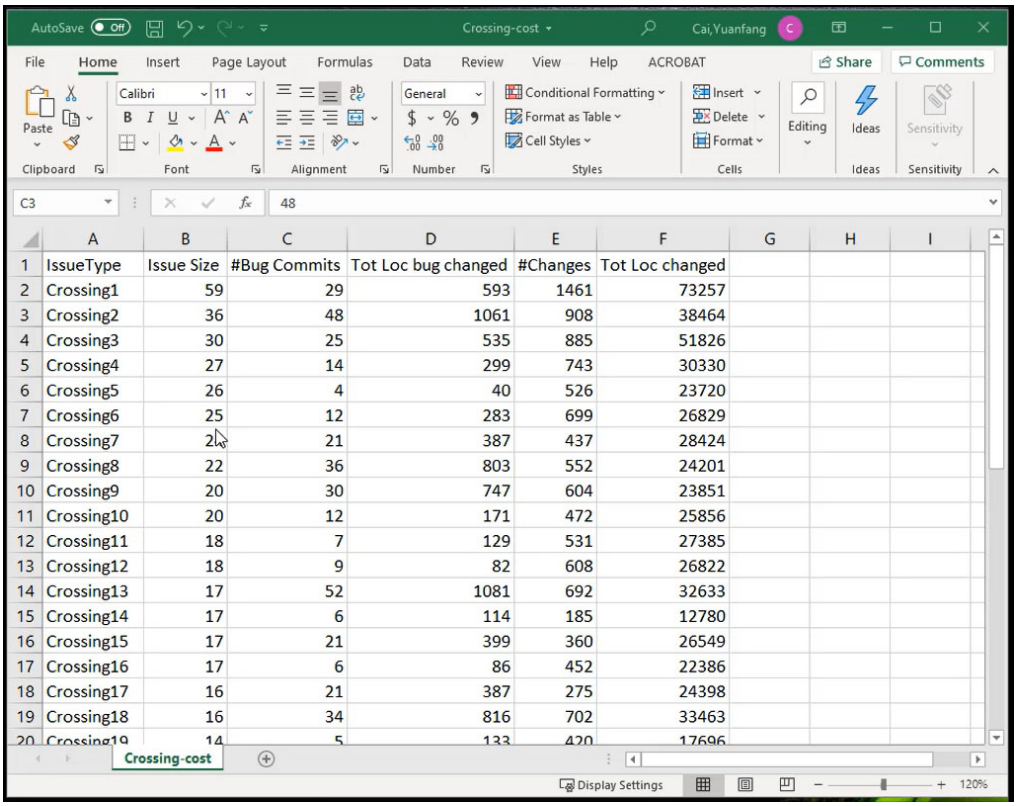
In the above example, the 9th row summarizes the overall project data. In this example, the project has 1304 files in total.
As recorded in its revision history, there are 173 commits related to bug fixes;
These bug fixing revisions consumed 4541 lines of code.
In total, there are 11,159 commits in the given time period, consuming 568465 line of code, including both bug related changes and other changes. <break time="1s"/>

Now let look at the first row that summarizes the maintenance costs of the Crossing anti-pattern:
There 289 files involved in this anti-pattern,
which represents 22.2% of all files.

There are 95 bug-related commits on these 289 files,
which represents 54.9% of all bug-related commits.
There are 1931 lines of code changed to fix bugs within these files, which consume 42.5% of the total.
Similarly, these 289 files are involved in 5,232 commits,
46.9% of the total.

The commits in this one Crossing consumed 254818 lines of code, or 44.8% of the project total.
This 2nd row shows that the Crossing anti-pattern has a a significant impact in the system's maintenance costs.

- Other csv files summarized the maintenance costs of other types of anti-patterns in a similar way



	A	B	C	D	E	F
1	IssueType	Issue Size	#Bug Commits	Tot Loc bug changed	#Changes	Tot Loc changed
2	Crossing1	59	29	593	1461	73257
3	Crossing2	36	48	1061	908	38464
4	Crossing3	30	25	535	885	51826
5	Crossing4	27	14	299	743	30330
6	Crossing5	26	4	40	526	23720
7	Crossing6	25	12	283	699	26829
8	Crossing7	24	21	387	437	28424
9	Crossing8	22	36	803	552	24201
10	Crossing9	20	30	747	604	23851
11	Crossing10	20	12	171	472	25856
12	Crossing11	18	7	129	531	27385
13	Crossing12	18	9	82	608	26822
14	Crossing13	17	52	1081	692	32633
15	Crossing14	17	6	114	185	12780
16	Crossing15	17	21	399	360	26549
17	Crossing16	17	6	86	452	22386
18	Crossing17	16	21	387	275	24398
19	Crossing18	16	34	816	702	33463
20	Crossing19	14	5	133	420	17696

For example, the above spreadsheet summarizes the costs of each Crossing instance, sorted by the “Issue Size”, that is, the number of files involved in the instance. The first instance involves 59 files, but it has fewer bug commits than the second crossing. The user can sort the spreadsheet using different measures, and can open the DSM of each instance.

- The anti-pattern-instances folder: this folder contains the dsms of each instances of each type, which can be opened using the DV8 GUI.

dsm

This folder contains the following design structure matrices:

- A matrix containing only structural dependencies
- A matrix containing only the co-change relationships between file pairs
- And a combined structural and co-change matrix

This directory also contains two clustering files. One file is clustering according to the package structure. The other is the design rule hierarchy clustering. Users can use DV8's matrix viewer to open any dependency matrix, and each matrix can be clustered in various ways.

hotspot

If the user does not provide a target issue id list, then the “seed” spreadsheets contain information about all files changed by more than two different commits.

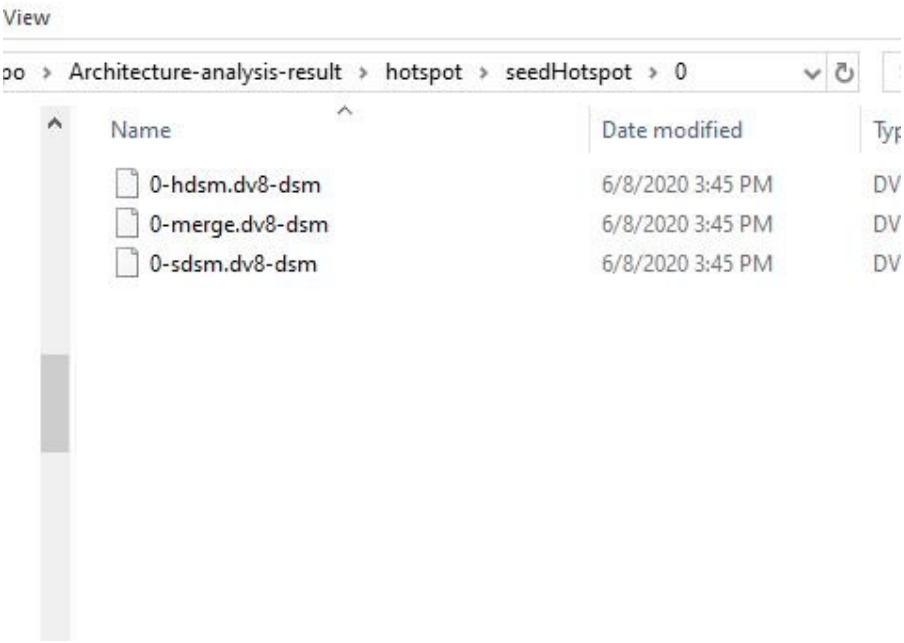
In DV eight, hotspots are defined as groups of files that are frequently modified or bug-prone within a given time period. DV8 extracts the design relationship between these files, so that users can analyze whether the high maintenance cost of these files is caused by design defects.

After executing the "Analysis Software" function, or the ark-report command in the DV eight console, DV eight stores all analysis results in the "Hotspot" folder. The user can also use “hotspot” and “hotspot cost” from the DV eight console to detect active hotspots and their associated maintenance costs within a specified period of time.

Now we elaborate on the hotspot analysis results.

The *seed-group* folder contains a list of files that have been modified for various reasons. If the user's input contains a list of issue IDs, such as bug ticket IDs, this file lists the file groups that have been modified by multiple bugs; If the user's input does not include a target ID list, this file lists the file groups that have been modified by multiple different commits. By default, DV8 considers files that have been modified because of two or more bug issues as seed files. The user can modify this threshold using DV 8 console commands. In this sample project, 7 files were modified due to multiple bug fixes. We call these files “seeds” because they usually violate the single responsibility principle and are core files with design flaws. These seed files frequently propagate errors to multiple other files.

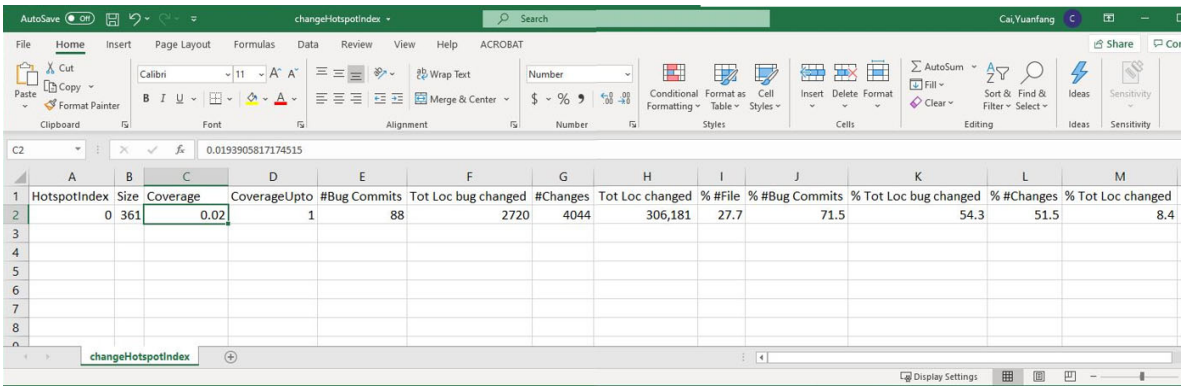
The *seed-hotspot* folder contain the DSMs composed of these core files only. In the sample project shown below, DV8 found only one hotspot, indicating that these 7 files are related in design. This phenomenon is consistent with our research results: the more error-prone or change-prone a file is, the more likely it relates to other files in the architecture.



For each hotspot, Dv eight automatically generates three matrices: a matrix that contains only structural information (*-sdsm.dv8-dsm), a matrix that contains only co-change information (*-hdsm.dv8-dsm), and a matrix that combines the two (*-mergedv8-dsm). Users can open these files with the DV8 viewer and observe their relationships.

In order to analyze the impact of these core files on other files, DV eight also generated a set of Change Hotspots, including all files that have design relations with the core files. In this sample project, there are 361 other files that are connected with these 7 core files.We can open this matrix and analyze the design anti-patterns within it. These 361 files exhibit all six anti-patterns.

These index files list the basic information of hotspots and their maintenance costs. The following form displays the information of this hotspot with 361 files.



	A	B	C	D	E	F	G	H	I	J	K	L	M
1	HotspotIndex	Size	Coverage	CoverageUpto	#Bug Commits	Tot Loc bug changed	#Changes	Tot Loc changed	% #File	% #Bug Commits	% Tot Loc bug changed	% #Changes	% Tot Loc changed
2	0	361	0.02	1	88	2720	4044	306,181	27.7	71.5	54.3	51.5	8.4

The data in column C is the proportion of files with high bug rates in this file group. Among these 361 files, about 2% of the files showed high bug rates.

The data in column D is the proportion of all the project's buggy files covered by the file group. Cell D2 is 100%, indicating that all files in the system that are bug-prone are contained in this file group.

Columns E, F, G, and H respectively list the number of bugs, the number of lines of code modified for bug-fixes, the number of modifications, and the lines of modified code in this file group.

Columns I, J, K, L, M show the proportions of these values to the total maintenance cost.

These data show that although this hotspot only involves about 28% of the files in the system, it is responsible for about 72% of all bug-related changes, and 54.3% of all bug-related lines of code.

Analyzing these two hotspots as design debt, these two ROI spreadsheets quantify their debt, and calculate the expected return on investment after refactoring.

root

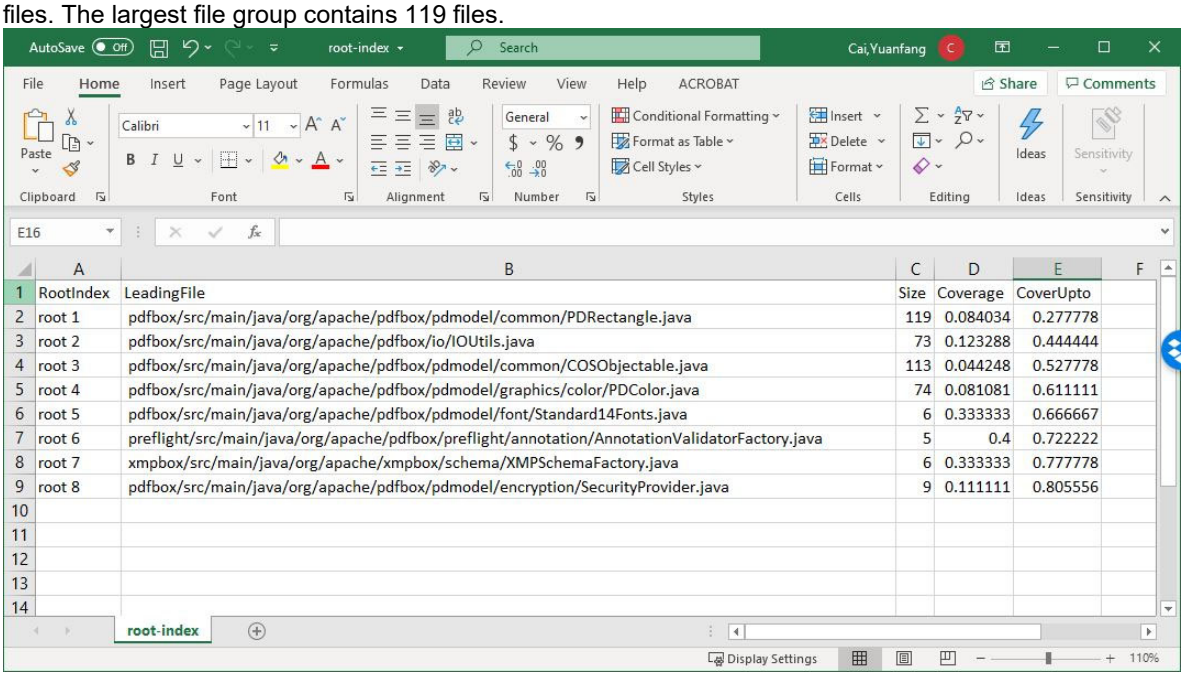
DV8 automatically calculates a set of correlation matrices covering files that are error-prone or change-prone. The objective is to reveal design issues that lead to high maintenance costs. Studies have shown that 50% to 90% error-prone files will be concentrated in 5 or fewer file groups. The more error-prone a file is, the more likely that it is connected with other files, resulting in the spread of defects in multiple files. We call these file groups that cover most error-prone files as **Root** spaces.

The root folder generated by DV8 a *root-spaces* subfolder containing all root file groups. Taking these file groups as design debt, the *root-roi.csv* file quantifies their debt and calculates the return on investment (ROI).

The root folder contains the design structure matrix of each file group, which can be opened with the DV8 viewer, so that the user can check the design anti-pattern within it.

The *root-index.csv* file summarizes the information and error-prone, or change-prone coverage of each file group. If the user's input contains a list of target issue ticket IDs, such as bug ticket IDs, these automatically generated file groups covers 80% of error-prone files; If the user's input does not include a target ID list, these file groups covers 80% of change-prone files. By default, DV8 considers files that are changed twice or more for bug-fixing as error-prone. The user can change these thresholds using the "arch-root" command line.

The following sample spreadsheet contains the information of 8 root file groups, sorted by the number of files. The largest file group contains 119 files.



	A	B	C	D	E	F
1	RootIndex	LeadingFile	Size	Coverage	CoverUpto	
2	root 1	pdfbox/src/main/java/org/apache/pdfbox/pdmodel/common/PDRectangle.java	119	0.084034	0.277778	
3	root 2	pdfbox/src/main/java/org/apache/pdfbox/io/IOUtils.java	73	0.123288	0.444444	
4	root 3	pdfbox/src/main/java/org/apache/pdfbox/pdmodel/common/COSObjectable.java	113	0.044248	0.527778	
5	root 4	pdfbox/src/main/java/org/apache/pdfbox/pdmodel/graphics/color/PDColor.java	74	0.081081	0.611111	
6	root 5	pdfbox/src/main/java/org/apache/pdfbox/pdmodel/font/Standard14Fonts.java	6	0.333333	0.666667	
7	root 6	preflight/src/main/java/org/apache/pdfbox/preflight/annotation/AnnotationValidatorFactory.java	5	0.4	0.722222	
8	root 7	xmpbox/src/main/java/org/apache/xmpbox/schema/XMPSchemaFactory.java	6	0.333333	0.777778	
9	root 8	pdfbox/src/main/java/org/apache/pdfbox/pdmodel/encryption/SecurityProvider.java	9	0.111111	0.805556	
10						
11						
12						
13						
14						

The file in column B is the core file of the group, which means that other files directly or indirectly depend on it.
The data in column D is the proportion of files that are either error-prone, or change-prone. For example, in the first file group, about 8% of the files appear to be error-prone.
The data in column E is the proportion of error-prone or change-prone files covered by multiple file groups.

For example, the first file group covers 28% error-prone files; The first and second file groups together cover 44% of all error-prone files; The first three file groups cover more than 80% of error-prone files.

The user can also open a root file in a DV8 viewer, detect the anti-patterns within a root, and further analyze the design problems that lead to high bug rate and high change rate.

Knowledge Base

In this section, we introduce the key concepts and the unique research advances that form the foundation of DV8, including:

- Design Structure Matrix (DSM) and Design Rule Hierarchy (DRH)
- Maintainability Metrics Suite
- Architecture Anti-patterns
- Architecture Roots
- Architecture Debt Quantification

Design Structure Matrix and Design Rule Hierarchy

The architectural analysis techniques provided in DV8 are mainly based on the theoretical foundations provided in Baldwin and Clark's Design Rule Theory [1]. In their book they state that software should be structured by design rules and independent modules. In a software system, design rules are often manifested as the important design decisions, which decouple the rest of the system into independent modules. A design rule is typically manifested as an interface or abstract class. For example, if an Observer Pattern [2] is used in a code base, then there must exist an observer interface that decouples the subject and concrete observers into independent modules. As long as the interface is stable, addition, removal, or changes to concrete observers should not influence the subject. In this case, the observer interface is

considered to be a *design rule*, decoupling the subject and concrete observers into two independent modules. Consider another example: if a Strategy Pattern [2] is implemented, then the strategy interface is considered as the design rule which decouples the context and concrete strategies into independent modules.

Design Rule Hierarchy (DRH). To automatically identify the design rules and the files that they influenced, our prior work introduced a clustering algorithm— Design Rule Hierarchy (DRH) [3], [4], [5], which clusters the files of a system into a hierarchical structure. Within such a hierarchy, files in layer L_i should only depend on files in the higher layers, L_{i-1} to L_1, and files in the top layer, L_1, should not depend on files in the lower layers, L_{i+1} to L_n. Hence files in the top layer, L_1, should contain the most influential interfaces or abstract classes, which do not depend on files in other layers. In addition, files in the same layer should be decoupled into a set of modules that are mutually independent from each other. Thus the changes, addition, even replacement to a module will not influence other modules within the same layer. Thus, the independent modules in the bottom layer of a design rule hierarchy are the most valuable, from an evolutionary perspective, because changes to these modules will not affect the rest of the system.

Design Structure Matrix (DSM). In DV8, the basic model to represent and visualize relationships between files is the Design Structure Matrix (DSM). A DSM is a square matrix, in which rows and columns are labeled with names of important system entities—in our case file names—in the same order. An annotation in the cell in row x, column y, $cell(r_x, c_y)$, indicates that there is a dependency relation between file x and file y: file x either structurally depends on file y, or file x and file y were changed together as recorded in the project's revision history.

The DSM in Fig. 1 presents a design rule hierarchy (DRH) with 3 layers: $L_1:(rc_1 - rc_2)$, $L_2:(rc_3 - rc_{11})$, $L_3:(rc_{12} - rc_{32})$. The first layer, L_1, contains the most influential design rules that should remain stable. Files in L_2 only depend on files in L_1. Similarly, files in L_3 only depend on files in the first two layers. Within each layer, files are grouped into mutually independent modules. Taking the bottom layer L_3 as an example: it is grouped into 8 mutually independent modules: $M_1:(rc_{12})$, $M_2:(rc_{13} - rc_{16})$, $M_3:(rc_{17} - rc_{18})$, etc. We can observe, from the absence of annotations in the cells shared by these modules, that there are no dependencies between them. The text in a cell is used to indicate specific types of dependencies between the files. For example, $cell(r_4, c_1)$ in Fig 1 is marked with "dp", which means `ExpressionBuilder.java` "*depends on*" (calls methods from) `ExpressionDefinition.java`.

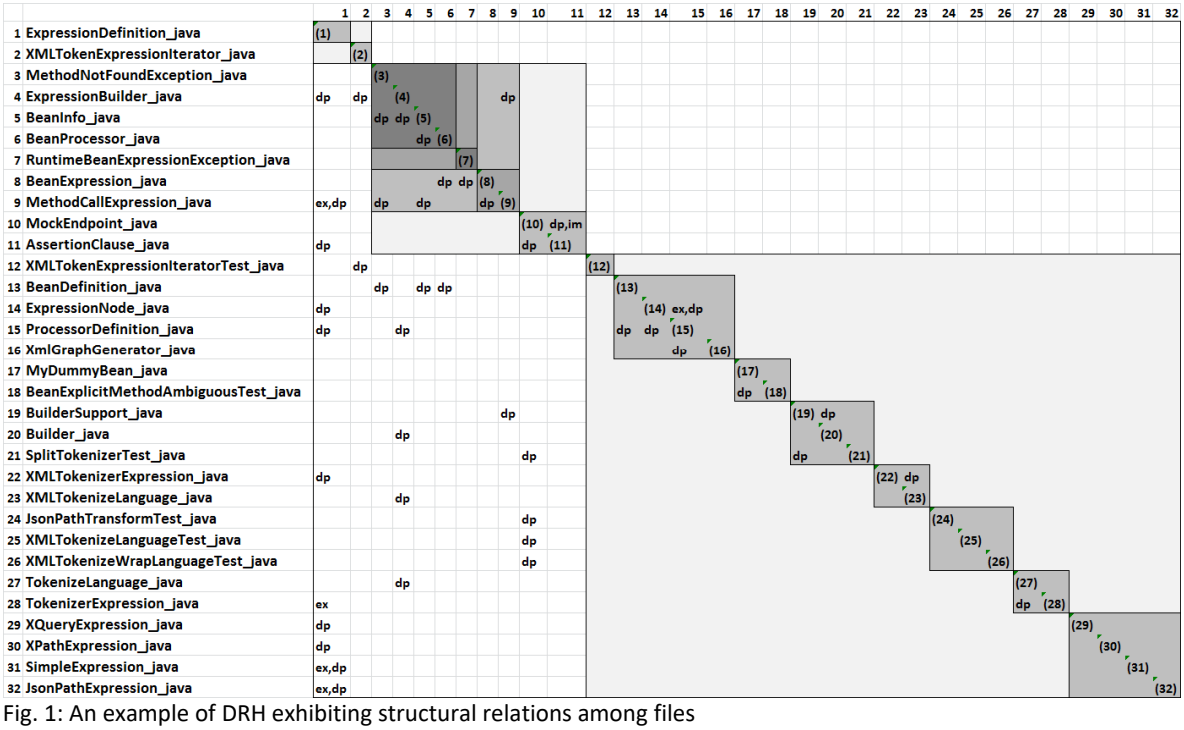


Fig. 1: An example of DRH exhibiting structural relations among files
ex: Extend; im: Implement dp: Depend

As we mentioned before, a DSM can also represent evolutionary coupling between files, i.e., the number of times two files were changed together. In Fig 2, a cell with just a number means that there is no structural relation between these two files, but they have been co-committed. For example, `cell(r8, c3)` is only marked with "*4*", which means that there is no structural relation between the files `BeanExpression.java` and `MethodNotFoundException.java`, but they have been changed and committed together 4 times, according to the project's revision history. A cell with both a textual annotation and a number means that the two files have both structural and evolutionary coupling relations. For example, `cell(r22, c1)` is marked with "*dp; 3*", which means that `XMLTokenizerExpression.java` depends on `ExpressionDefinition.java`, and they were changed together 3 times.

.

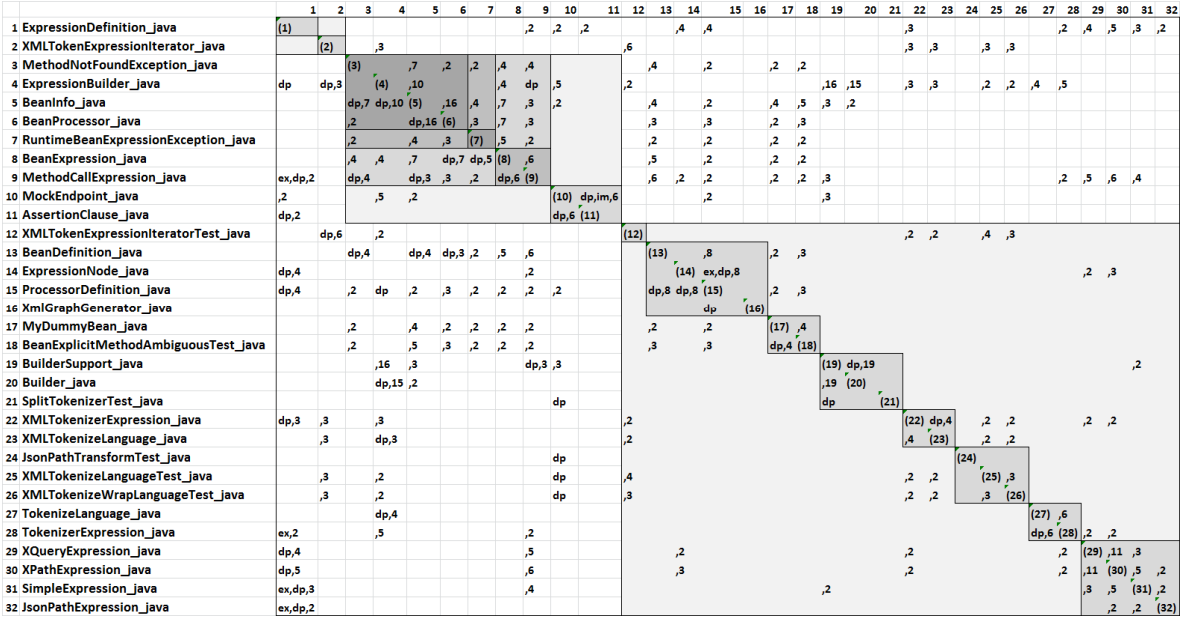


Fig. 2: An example of DRH exhibiting structural relations and evolutionary coupling among files
ex: Extend; im: Implement dp: Depend

[1] C. Y. Baldwin and K. B. Clark, **Design Rules, Vol. 1: The Power of Modularity**. *MIT Press*, 2000.
[2] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, **Design Patterns: Elements of Reusable Object-Oriented Software**. *Addison-Wesley*, 1994.
[3] Y. Cai and K. J. Sullivan, **Modularity analysis of logical design models**, in *Proc. 21st IEEE/ACM International Conference on AutomatedSoftware Engineering*, Sep. 2006, pp. 91–102.
[4] S.Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi, **Design rule hierarchies and parallelism in software development tasks**, in *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 197–208.
[5] Y. Cai, H. Wang, S. Wong, and L. Wang, **Leveraging design rules to improve software architecture recovery**, in *Proc. 9th International ACM Sigsoft Conference on the Quality of Software Architectures*, Jun. 2013, pp. 133–142.

Maintainability Metrics Suite

DV8 provides a state-of-the-art maintainability metrics suite, including the following metrics:

Decoupling Level (DL) [1] measures how well an architecture is decoupled into modules base on the Design Rule Hierarchy (DRH) clustering. If a module influences all other files directly or indirectly in lower layers, its DL is 0—this is the worst case where a system's files are fully connected. And if a system's files had no dependencies on each other its DL would be 1. All real systems fall somewhere within this range. The more files a given file influences in lower layers, the lower its DL. In addition, the larger a module, the more likely it will influence more files in the lower layers, and hence the lower its DL. Conversely, the more that files in a lower layer are *independent* from files in upper layers, the higher the DL.

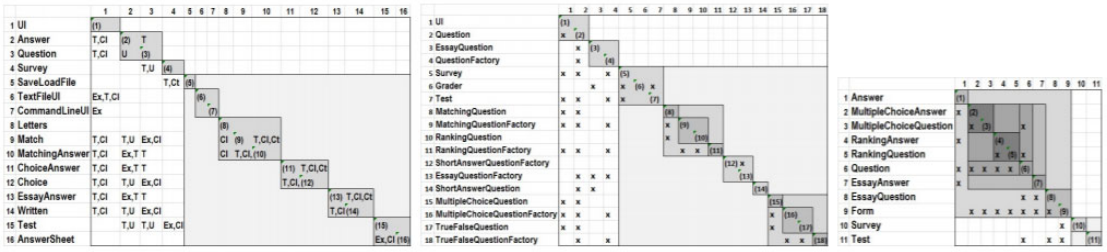
Propagation Cost (PC) [3] was defined by MacCormack et al. PC is calculated based on a DSM representation of a system's dependencies and aims to measure how tightly coupled a system is. Given a

DSM of a project's files and their dependencies, the algorithm first calculates its transitive closure to add indirect dependencies to the DSM until no more can be added. Given the final DSM with all direct and indirect dependencies, PC is calculated as the number of non-empty cells divided by the total number of cells. For example, the PCs of the three DSMs in Figure 1 are 25%, 37%, and 51% respectively. The lower the PC, the less coupled the system.

The limitation of PC is that it is sensitive to the size of the DSM: the greater the number of files, the smaller the PC. For example, from the 46 open source projects with more than 1000 files, 70% of them have PCs lower than 20%. For the other 62 projects with less than 1000 files, however, about 48% of them have PCs lowers than 20%. More importantly, an architecture can change drastically without significantly changing its PC.

Independence level [IL] [4] proposed in our prior work. Based on design rule theory [5], the more independent modules there are in a system, the higher its option value. In our prior work [4], we proposed a metric called Independence Level (IL) to measure the portion of a system that can be decoupled into independent modules within the last layer of its DRH. For example, the IL in the DSM of Figure 1a is 0.75 because 12 out of the 16 files are in the last layer. The Decoupling Level metric we propose here improves on the IL metric.

The limitation with IL is that it doesn't consider the modules in the top layers of a DRH, nor does it consider the size of a module. It is observed that there are cases where the lowest layer contained very large modules. In these cases, even through the IL appeared to be high, the system was not well modularized. In other cases, we observed that even though the number of files decoupled in the last layer were not large, the modules in upper layers had few dependents. In this case, a system may not experience maintenance problems, despite its low IL.



(a) Submission 1: DL = 82%; PC = 25%; IL = 75%

(b) Submission 2: DL = 78%; PC = 37%; IL = 78%

(c) Submission 3: DL = 18%; PC = 51%; IL = 18%

Figure 1: Design Rule Hierarchy Samples. T:*Typed*; Cl:*Call*; Ex:*Extend*; Ct:*Cast*, U:*Use*, x: any dependency

Please refer to the Measure Modularity section that explains how to use DV8 to assess a software using this metrics suite.

[1] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao, Qiong Feng: **Decoupling level: a new metric for architectural maintenance complexity**. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE 2016, Page 499-510.
[2] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. **Design rule hierarchies and parallelism in software development tasks**. In *Proceedings 24th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2009, pages 197–208, Nov. 2009.
[3] A. MacCormack, J. Rusnak, and C. Y. Baldwin. **Exploring the structure of complex software designs: An empirical study of open source and proprietary code.** *Management Science*, 52(7):1015–1030, July 2006.
[4] K. Sethi, Y. Cai, S. Wong, A. Garcia, and C. Sant'Anna. From **retrospect to prospect: Assessing modularity and stability from software architecture.** *In Proceedings of the 8th Working IEEE/IFIP International Conference on Software Architecture*, WICSA 2009, Sept. 2009.
[5] C. Y. Baldwin and K. B. Clark. Design Rules, Vol. 1: **The Power of Modularity**. MIT Press, 2000.

Design Anti-patterns

Using DV8, you can detect 6 types of architecture anti-patterns as defined in [1]. (Note that, in DV8, they are also referred to as *architecture issues.*) These anti-patterns were defined based on Baldwin and Clark's design rule theory [2] and violation of prevailing design principles, such as the famous SOLID principle.

The following anti-patterns can be detected using structural information only:

- Clique: a group files that are interconnected, forming a strongly connected “component” but not belonging to a single module.
- Package Cycle: a group of packages that depend on each other in a cyclic relation
- Improper Inheritance: an inheritance hierarchy that violates the Liskov Substitution Principle.

The following three anti-patterns can only be detected with both structural relation and co-change information:

- Unstable Interface: file(s) that have a large number of dependents and change frequently with these dependents.
- Crossing: a file with both high fan-in and high fan-out and which changes frequently with all of its relationships—the set of files it depends on and the set of files that depend on it.
- Modularity Violation: a group of files that do not have structural relationships, but which change frequently with each other.

The rationale, description, and visualization of these anti-patterns are elaborated in the following subsections.

Please refer to the Detect Architecture Anti-pattens section that explains how to use DV8 to detect these anti-patterns.

[1] Ran Mo, Yuanfang Cai, Rick Kazman, Lu Xiao: **Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells.** In *Proceedings of 12th Working IEEE/IFIP Conference on Software Architecture,* WICSA 2015: 51-60
[2] C. Y. Baldwin and K. B. Clark. Design Rules, Vol. 1: **The Power of Modularity**. MIT Press, 2000.

Clique

`Rationale:` It is widely accepted that cyclic dependencies should be avoided. To reduce the number of instances you have to examine, we define Clique as a set of files whose structural relations form a strongly connected graph, so that changes to any files can be propagated to any other files within the group.

`Description:` If there is a subset of files that form a strongly connected component based on their structural relations, this file group is identified as a Clique instance.

`Visualization`: In a DSM, the dependencies that form cycles can be shown as symmetric, non-empty cells, as shown below:

		1	2	3	4	5	6	7	8	9	10	11	12
1	ActivityRules_java	(1)	,5	dp,3	,3	,3	,4	,2	,2	,2	dp,4	dp,3	,3
2	ProcessInstance_java	dp,5	(2)	,3	dp,11	,4	,4	,6	,6	,2	,3	,8	,4
3	ProcessRules_java	dp,3	,3	(3)	,2		,3	,2	,2	,2	,3	,2	,2
4	ActivityState_java	dp,3	dp,11	,2	(4)	,3	,3	,7	,6	dp,2	,3	,7	,2
5	JpaBamProcessorSupport_java	dp,3	,4	dp	,3	(5)	,4	,2	,2	,2	,3	,6	,7
6	JpaBamProcessor_java	dp,4	dp,4	dp,3	dp,3	dp,4	(6)	,3	,3	dp,2	,4	,3	,5
7	TimeExpression_java	dp,2	dp,6	,2	,7	,2	,3	(7)	dp,8	,2	dp,3	,7	,2
8	ActivityBuilder_java	dp,2	,6	,2	,6	,2	,3	,8	(8)	,2	,3	dp,10	,2
9	ProcessContext_java	dp,2	dp,2	,2	dp,2	,2	,2	,2	,2	(9)	,2	,3	,2
10	TemporalRule_java	,4	,3	,3	dp,3	,3	,4	dp,3	dp,3	,2	(10)	,3	,4
11	ProcessBuilder_java	,3	dp,8	dp,2	,7	,6	dp,3	,7	dp,10	,3	,3	(11)	dp,5
12	ActivityMonitorEngine_java	,3	,4	dp,2	dp,2	,7	,5	,2	,2	,2	,4	,5	(12)

Fig.: An example of Clique: the highlighted cells are symmetric.
dp: depend

The figure above shows an instance of Clique. Files in this example are highly coupled with each other through multiple dependency cycles, such as, `ActivityRules.java <-> Process-Rules.java`, `ActivityRules.java -> TimeExpression java -> TemporalRule.java -> ActivityRules.java`, etc.

If you detect Clique anti-patterns using DV8, you can visualize each instance using DV8 Explorer: Select "`Analysis-> Load Issue File`", and choose a ".dv8-issue" file. For a Clique issue, DV8 will automatically arrange the files so that dependencies that form cycles are arranged into symmetric cells, as shown above.

Package Cycle

`Rationale:` Ideally, the package structure of a software system should form a hierarchical structure. As with the Clique anti-pattern, a cycle among packages reduces the understandability and maintainability of a system.

`Description:` Given two packages `Pa, Pb`, there exists a file `f1` in `Pa` and a file `f2` in `Pb`. Given another file `fj` in `Pb` and `fi` in `Pa`, if `f1` depends on `fj`, and `f2` depends on `fi`, then we consider that these two packages create a Package Cycle。

`Visualization`: In a DSM, the dependencies that form cycles among packages can be shown as symmetric, non-empty cells between packages as shown below:

		1	2	3	4
1	mapred.AvroOutputFormat_java	(1)			dp
2	mapred.FsInput_java		(2)		
3	file.SortedKeyValueFile_java		dp	(3)	
4	file.HadoopCodecFactory_java				(4)

Fig.: An example of Package Cycle: the two files causes the two packages to form cyclical dependencies.
dp: depend

The above figure presents shows an instance of Package Cycle, in which `AvroOutputFormat.java` in package `mapred` depends on `HadoopCodecFactory.java` in package `file`, and `SortedKeyValueFile.java` in package `file` depends on `FsInput.java` in package `mapred`, forming a dependency cycle between package `mapred` and package `file`.

If you detect Package Cycle anti-patterns using DV8, you can visualize each instance using DV8 Explorer: Select "`Analysis-> Load Issue File`", and choose a "`.dv8-issue`" file. For a Package Cycle instance, DV8 will automatically cluster the system based on namespaces, and arrange the packages so that file dependencies that form package cycles are manifested, as shown above. .

Improper Inheritance

Improper Inheritance Anti-pattern:

`Rationale:` According to our research, there are two most frequently observed problems in the implementation of inheritance hierarchy are: (1) a parent class depends on one of its children; and (2) a client class of the hierarchy depends on both the base class and its children.
Both cases violates Liskov Substitution principle [], since the parent class can no longer be a placeholder substitutable by any of its children. They also violates the Design Rule Theory [] because the parent class cannot be a decoupling design rule. They violate the Dependency Inversion Principle [18] since a client should depend on abstractions, not on concretions.

`Description`: DV8 detects an inheritance hierarchy to be problematic if it falls into one of the following two cases:
1) Given an inheritance hierarchy containing one parent file, `f_p`, and there exists a child file `f_c` in which `f_p` depends on `f_c`;
2) Given an inheritance hierarchy containing one parent file, `f_p` , with one or more childern, there exists a client `f_client` of the hierarchy, that depends on both the parent and one or more of its children.

`Visualization`:

		1	2	3	4	5	6	7	8	9
1	ProcessorDefinition_java	(1)	dp,8			,2	,2		,6	,3
2	AggregateDefinition_java	ex,8	(2)							
3	JmsEndpoint_java			(3)	dp,10		,2	,2	,2	
4	JmsQueueEndpoint_java			ex,10	(4)					
5	ManagedPerformanceCounter_java	,2				(5)	,3	,4	,5	
6	ManagedProcessor_java	dp,2		,2		ex,3	(6)	,7	,9	
7	ManagedCamelContext_java			,2		ex,4	,7	(7)	,21	,2
8	ManagedRoute_java	,6		,2		ex,5	,9	,21	(8)	
9	DefaultManagementObjectStrategy_java	,3				dp	dp	dp,2	dp	(9)

Fig.: Instances of Unhealthy Inheritance Hierarchy architecture
dp: depend, ex: extend

The figure above presents several instances of Unhealthy Inheritance Hierarchy: 1) the parent file, `ProcessorDefinition.java` depends on its child file `AggregateDefinition.java`; 2) the parent file, `JmsEndpoint.java` depends on its child file `JmsQueueEndpoint.java`; 3) the client file `DefaultManagementObjectStrategy.java` depends on the parent file `ManagedPerformanceCounter.java` and all of its children.

You can detect and visualize each instance of Improper Inheritance using DV8 Explorer: Click "`Analysis-> Load Issue File`", and select a ".dv8-issue" file. For a Improper Inheritance issue, DV8 will automatically arrange the files so that the inheritance hierarchy is arrange at the top of the DSM, and the clients depend on both the parent and children were highlight, as shown below:

Modularity Violation

Modularity Violation Anti-pattern:

`Rationale`:

Baldwin and Clark's Design Rule theory [17] proposed that independent modules can be changed or even replaced without influencing each other. Wong et al. introduced the term Modularity Violation [26], which describes two structurally independent modules that change together
frequently, meaning that they are not truly independent. The more often two structurally unrelated files change together, the more likely that there are implicit dependencies between them [26], [27]. In this paper, we calculate the minimal number of file groups with modularity violations.

`Description`:

A Modularity Violation Group (MVG) contains a set of modularity violation files. We calculate the minimal number of MVGs so that their union covers all violated file pairs (two files without structural relations but changed together) in a project. In a Modularity Violation Group, there
exists a core file, fcore, which all other files are not structurally related to, but have frequently changed together with. To identify a Modularity Violation Group (MVG), our tool first generates all filesets by considering each file in a project as a core file, then greedily searches a fileset that

covers most violated file pairs as a MVG, until the union of all the MVGs covers all violated file pairs in a project.

`Visualization:`

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	**DropIndexStatement_java**	(1)	14	13	12	10	8	7	6	6	5	5	5	5	5	5
2	**CreateIndexStatement_java**	14	(2)	16	11	10	8	7	6	6	12	7		4	5	6
3	**AlterTableStatement_java**	13	16	(3)	12	9	7	8	6	6	12	9	3	6	5	9
4	**CreateKeyspaceStatement_java**	12	11	12	(4)	12	7	10	5	5	4	4		3	5	6
5	**DropKeyspaceStatement_java**	10	10	9	12	(5)	4	7	5	5	3	3		3	5	5
6	**CassandraServer_java**	8	8	7	7	4	(6)				21	24	43			
7	**AlterKeyspaceStatement_java**	7	7	8	10	7		(7)	5	5			3	3	5	5
8	**DropTriggerStatement_java**	6	6	6	5	5		5	(8)	12				4	5	5
9	**CreateTriggerStatement_java**	6	6	6	5	5		5	12	(9)				4	5	5
10	**SelectStatement_java**	5	12	12	4	3	21				(10)	48	8			3
11	**ModificationStatement_java**	5	7	9	4	3	24				48	(11)				3
12	**StorageService_java**	5		3			43	3			8	0	(12)			
13	**AlterTypeStatement_java**	5	4	6	3	3		3	4	4				(13)	3	4
14	**DropTableStatement_java**	5	5	5	5	5		5	5	5				3	(14)	5
15	**CreateTableStatement_java**	5	6	9	6	5		5	5	5	3	3		4	5	(15)

Fig. : An example instance of Modularity Violation Group
Each number indicates the co-changes between two files

The above figure presents an instance of MVG detected in Apache Cassandra. There are no structural dependencies between `DropIndexStatement.java` and the other files. However, the cells annotated with a number in the DSM reveal that all other files changed together at least 3 times with `DropIndexStatement.java`, the core file.

You can detect and visualize each instance of Modularity Violation using DV8 Explorer: Click "`Analysis-> Load Issue File`", and select a ".dv8-issue" file. For a Modularity Violation issue, DV8 will automatically arrange the files so that the core files is arrange at the top of the DSM,

Crossing

Crossing Anti-pattern:

`Rationale:`

If a file has both a large number of dependents and depends on a large number of other files, i.e., with both high fan-in and high fan-out, it is unlikely that this file follows Single Responsibility Principle [18]. We observe that if such a file also changes frequently with its dependents and the files it dependents on, it is often the center of error- and change-proneness.

`Description:`

If a file is changed frequently with its dependents and the files that it depends on, then we consider these files to follow a Crossing anti-pattern (CRS).

`Visualization:`

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	ErrorHandlerBuilderRef_java	(1)	,2		,6			x,7			x,4		,3		x,9				
2	BuilderSupport_java	,2	(2)		x,10						x,2			,5	x,2				
3	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelayTest_java		x	(3)		,2					x,2			,2			,3	,2	,3
4	DeadLetterChannelBuilder_java	,6	,10		(4)		,11	x,5		,2	x,14		,8	x,10	x,8				
5	RedeliveryErrorHandlerNonBlockedDelayTest_java		x	,2		(5)					x,2			,2			,2	,2	,2
6	DefaultErrorHandler_java				,11		(6)			,2	,10		,5	x					
7	ErrorHandlerBuilderSupport_java	,7			,5			(7)			,4		,2		x,10				
8	RedeliveryErrorHandlerNoRedeliveryOnShutdownTest_java		x						(8)		x,2			,2					
9	CamelErrorHandlerFactoryBean_java				,2		,2			(9)	x,2			x	x				
10	**DefaultErrorHandlerBuilder_java**	,4	,2	,2	,14	,2	x,10	x,4	,2	,2	(10)	,2	,6	x,13	x,4	,2	,2	,2	,2
11	TransactionalClientDataSourceRedeliveryTest_java		x								x,2	(11)							
12	TransactionErrorHandlerBuilder_java	,3			,8		,5	x,2			x,6		(12)	,2	,3				
13	RedeliveryPolicy_java		,5	,2	,10	,2			,2		,13		,2	(13)	,2		,2	,2	,2
14	ErrorHandlerBuilder_java	,9	,2		,8			,10			,4		,3	,2	(14)				
15	OnExceptionRouteWithDefaultErrorHandlerTest_java		x								x,2					(15)			
16	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay3Test_java		x	,3		,2					x,2			,2			(16)	,2	,3
17	RedeliveryErrorHandlerBlockedDelayTest_java		x	,2		,2					x,2			,2			,2	(17)	,2
18	AsyncEndpointRedeliveryErrorHandlerNonBlockedDelay2Test_java		x	,3		,2					x,2			,2			,3	,2	(18)

Fig. : An example of Crossing
x indicates structural dependencies, such as extend, depend, etc.

The above figure presents an instance of Crossing. We can see that the center file, `DefaultErrorHandlerBuilder.java`, was changed frequently with its dependents and the files it depends on in the revision history.

You can detect and visualize each instance of Crossing using DV8 Explorer: Click "`Analysis-> Load Issue File`", and select a ".dv8-issue" file. For a Crossing issue, DV8 will automatically arrange the files so that center is arrange in the middle of the DSM, and the violation files are highlighted.

Unstable Interface

Unstable Interface Anti-pattern:

`Rationale:`

According to the design rule theory [17] and design principles [18], important and influential abstractions (design rules) should be stable. Otherwise their bugs and changes can be propagated to multiple files. We have observed that unstable or poorly-designed abstractions are
often related to high-maintenance, and deserve special attention.

`Description:`

If a highly influential file (files with a large number of dependents) is changed frequently with other files as shown in the revision history, then we call it an Unstable Interface (UIF).

`Visualization:`

You can visualize each instance of Unstable Interface using DV8 Explorer: Click "`Analysis-> Load Issue File`", and select a ".dv8-issue" file. For a Unstable Interface issue, DV8 will automatically arrange the files so that the interface files are arrange at the top of the DSM, and the violations are highlighted

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	**StreamSession_java**	**(1)**	x,10	x,8	4	2	4	2	3	7	2	3	2	2	2	3	8	2	2
2	ConnectionHandler_java	**x,10**	(2)	2	x,3				3	5							4		
3	StreamTransferTask_java	**x,8**	2	(3)	3		x,4				2	3	2			2	5		
4	StreamMessage_java	**x,4**	3	3	(4)	x	x,2		2	2		3	2				2		
5	ReceivedMessage_java	**x,2**			x	(5)													
6	OutgoingFileMessage_java	**x,4**		4	x,2		(6)				2	2				2	3		
7	SessionInfo_java	**x,2**						(7)		2							2	2	2
8	StreamCoordinator_java	**x,3**	3		2				(8)	4				2			3		
9	StreamPlan_java	**x,7**	5		2			2	x,4	(9)							5	2	2
10	LegacySSTableTest_java	**x,2**		2			2			x	(10)						2		
11	StreamReader_java	**x,3**		3	3		2					(11)	4			6			
12	CompressedStreamReader_java	**x,2**		2	2							x,4	(12)			2			
13	StreamEvent_java	**x,2**						x	2					(13)					
14	StreamStateStoreTest_java	**x,2**												x	(14)				
15	StreamReceiveTask_java	**x,3**		2			2					6	2			(15)			
16	StreamTransferTaskTest_java	**x,8**	4	x,5	2		3	2	3	5	2						(16)	2	2
17	SessionInfoCompositeData_java	**x,2**						x,2		2							2	(17)	2
18	SessionInfoTest_java	**x,2**						x,2		2							2	2	(18)

Fig.: An example of Unstable Interface
x indicates structural dependencies, such as extend, depend, etc.

The above figure depicts an instance of Unstable Interface in the Cassandra project. An "`x`" in a cell indicates a structural dependency between the file on the row and the file on the column; a number represents the historical co-change frequency of these two files. We can see that multiple files structurally depend on `StreamSession.java` and that these files have changed together frequently with it as evidenced by the project's revision history.

You can detect and visualize each instance of Unstable Interface using DV8 Explorer: Click "`Analysis-> Load Issue File`", and select a ".dv8-issue" file. For an Unstable Interface issue, DV8 will automatically arrange the files so that interface file is arrange at the top of the DSM, and the influenced files are highlighted.

Design Hotspot

In DV8, hotspots are defined as groups of files that are frequently modified or bug-prone within a given time period. DV8 extracts the design relationship between these files, so that users can analyze whether the high maintenance cost of these files is caused by design defects.

- If the user's input does not include a target ID list, DV8 will detect a set of files have been modified by multiple different commits, and calculate a **Change** Hotspot.
- If the user's input contains a list of issue IDs, such as bug ticket IDs,
    - DV8 first detects a set of files that have been modified by multiple bugs, and name these files as a **Seed** file group. These given issue IDs are defined as **Target** issues. By default, DV8 considers files that have been modified because of two or more bug issues as seed files. The user can modify this threshold using DV8 console commands.
    - DV8 will calculate a **Seed** Hotspot that contain the DSMs composed of these seed files only. In order to analyze the impact of these core files on other files, DV eight also generated a set of **Change** Hotspots, including all files that have design relations with the seed files.

Research shows that 65% of code modifications are usually concentrated in fewer than 3 hotspots. The members of active hotspots may change during system evolution, but major hotspots will exist for a long time; an average of 24.6 months If there are design anti-patterns in active hotspots, they usually reflect real design debt. Continuous tracking and detection of the occurrence and growth of hot spots can reveal early design issues and can help prevent severe

architecture decay. Fixing design issues in hotspots can avoid many defects in multiple files.

Architecture Roots

Using DV8, you can detect a small group of architecturally related files involved in a selected set of issues, such as bug issues, or refactoring issues, which can be provided as a target list (sample) file. This file lists the number of times each file is changed for a particular type of issues. This function will enable you to examine the architecture relations among files with similar properties, such as error-proneness. We call the detected file groups as *architecture roots*. If the target issues are bug issues, you could call them *bug roots*; if the target issues include all change activities, you can call them *change roots*.

Our research [1] has shown that just five bug roots typically cover 50% to 90% of the most ***error-prone*** files in a system. This observation has been validated over dozens of industrial and open source software systems. The implication is that most error-prone files are architecturally connected; the more error-prone the files are, the more likely that they are architecturally connected and that errors propagate through the connections.

Here file error-proneness is determined by the number of times a file is involved in bug fixes. The more often a file is changed to fix bugs, the more error-prone it is. Using DV8, the user can specify a threshold for a file to be considered as error-prone when executing related commands. In our research, we used a threshold of 2 for error-proneness, that is, files that were changed for bug fixes two or more times are considered as error-prone.

(1) A sample root: capturing most change-prone files and their architecture flaws. The following figure presents an example of one detected change root in an industrial project [2]. In this figure, the "*CF*" column lists change frequency of each file; and the "*Top*" column lists the percentage ranking in terms of change-proneness of each file. For example, the file "*p4.F3*" in row 26 was changed 361 times, and it ranked the most change-prone (top 0.1%ile) among all 2,403 changed files in Proj_SS. 84% of the files in this root ranked in the top 10th percentile most change prone, and six out of the 31 files ranked in the top 1st percentile, which indicates that the root is a true maintenance hotspot. Files in this root are clustered into three design rules hierarchy layers: L_1: (rc_1-rc_{27}), L_2: (rc_{28}) and L_3: (rc_{29}-rc_{31}). Files in each layer are recursively clustered into independent modules. For example, files 10 - 26 are grouped into 5 modules, and these modules are structurally independent from each other.

From each root, you can detect architecture anti-patterns within it that may be responsible for the propagation of bugs. For example, in the figure below: 1) p1.F1, an unstable interface, is depended upon by most of the files, and most of these dependents have changed together with it frequently; 2) Multiple dependency cycles are identified, such as, p1.F 5 ↔p2.F 2, and p2.F 2 →p2.F 1 →p1.F 6 →p1.F 5 → p2.F 2; 3) p1.F1 depends on its child, which is Unhealthy Inheritance; 4) Many modularity violations are highlighted in red: structurally independent modules that have changed together frequently.

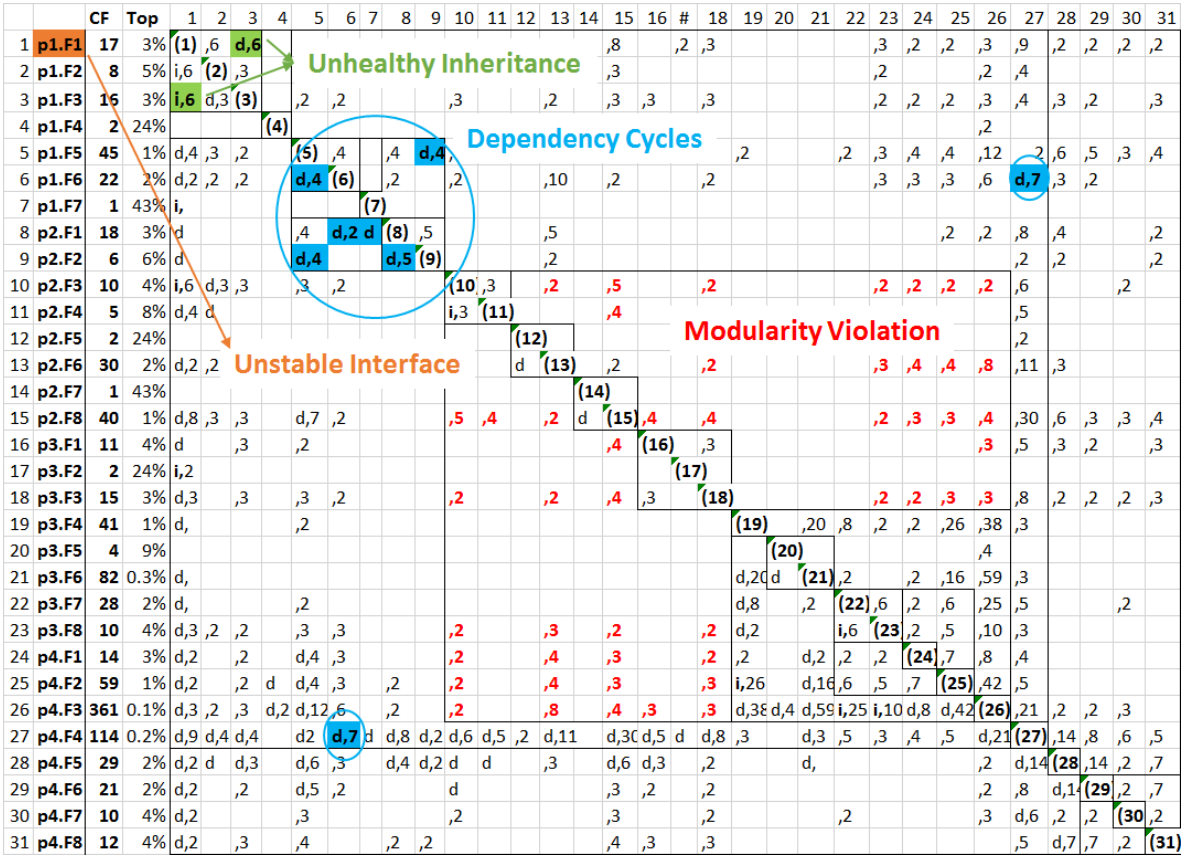


Figure: DRH-Clustered Architecture Root

`d: depend; i: inherit; CF: Change Frequency; Top: percentile rank`

(2) Cumulative effects of roots: a few roots capture most bugs or changes. The advantage of root detection is that you don't need to examine a large number of files or instances to figure out which architecture problems contribute most to error-proneness and/or change-proneness. Instead, you just need to examine a few, usually fewer than 5, file groups to figure out which architecture problems are most severe.

A file may participate in more than one root; that is, roots overlap with each other. DV8 also calculates their cumulative data, as shown in the following table (from [2]): In this table, "*Size*" means the number of distinct files in the first `n` roots, where, `n = 1, 2, ..., 4`. The "*%Size*" column presents the percentage of the root size compared with the total number of files in the project. For example, "*222*" in the second row means that `root1` and `root2` (the first 2 Roots) contain 222 distinct files, which cover 14% of all files in the project. The "*Coverage*" column presents the cumulative coverage of change-prone or bug-prone files by these roots. The fourth row of this table indicates all these 4 roots contain only 24% of all the files in this project, but cover 55% of all change-prone files and 65% of all bug-prone files. Files in each root are architecturally connected, hence change-proneness or bug-proneness may be propagated among these files.

			Coverage	
Root	Size	% Size	Change	Bug
root1	147	10%	24%	29%
root2	222	14%	38%	52%
root3	263	17%	47%	57%
root4	364	24%	55%	65%

Table: Cumulative Data of Architecture Roots

Please refer to the Detect Architecture Roots section that explains how to use DV8 to detect architecture roots.

[1] Lu Xiao, Yuanfang Cai, Rick Kazman: **Design rule spaces: a new form of architecture insight.** In *Proceedings of the 36th International Conference on Software Engineering* (ICSE 2014). Pages 967-977
[2] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, Martin Naedele: **Experiences Applying Automated Architecture Analysis Tool Suites.** In *proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (ASE 2018). Pages 779-789

Architecture Debt Quantification

Architecture roots, as well as, anti-patterns, can be considered as *architecture debt*, a type of technical debt (TD). The rationale is that, if these architecture problems are not fixed, they may continue to generate additional maintenance costs, the same way that a monatery debt accumulates interest. In DV8, you can calculate (1) the added maintenance costs due to each instance of each anti-pattern, and (2) the added maintenance costs of each architecture root.

(1) The maintenance costs of each instance of anti-patterns and roots: As an example, the following tables summarizes the anti-patterns detected in a real industrial project [1], their scopes and maintenance costs. The first line shows that there are 322 files (21% of all the files) involved in 26 Clique instances. These files were changed 1,790 times involving 26,294 LOC, 41% of all the LOC changed for the entire project. 643 of the changes are for bug fixing, involving 16,557 LOC, which is 45% of all the LOC spent for bug fixing. This table shows that Cliques are very expensive to maintain in this project. The table below shows that *Clique1* involves 99 files and incurred the most maintenance costs, definitely worth attention. *Clique5*, although it contains just 16 files, also appears to be very costly.

Using this table, you can prioritize which flaws need to be addressed in which order. By comparing with system average bug and change rates, we can see that files involved in these flaws are causing high maintenance difficulty.

Table: Architecture anti-patterns
Pt. : Percentage; Flaw CF - BC : maintenance costs, quantified by CF, CC, BF and BC, of the files in each flaw

	#Instances	#Files	Pt.	Flaw CF	Pt.	Flaw CC	Pt.	Flaw BF	Pt.	Flaw BC	Pt.
Clique	26	322	21%	1,790	28%	26,294	41%	643	34%	16,557	45%
Crossing	91	368	24%	3,146	50%	40,247	63%	1,051	55%	25,177	68%
ModularityViolation	667	588	38%	4,538	72%	46,224	72%	1,438	75%	27,648	74%
PackageCycle	175	499	32%	2,417	38%	29,906	47%	778	41%	18,889	51%
UnstableInterface	6	316	21%	1,669	26%	19,898	31%	388	20%	11,457	31%
UnhealthyInheritance	72	257	17%	1,528	24%	22,007	34%	480	25%	13,481	36%

Table: Maintenance costs of Clique instances

Instance Name	Size	Tot. CF	Tot. CC	Tot. BF	Tot. BC
Clique1	99	226	7,847	112	4,673
Clique2	78	181	431	7	212
Clique3	28	181	1,686	49	897
Clique4	18	246	3,130	39	1,427
Clique5	16	168	3,553	89	2,662

We can similarly calculate the maintenance costs incurred on each root, as exemplified in the following table. The first row shows that the first root involves 147 files. These files were changed 1,109 times, consuming 13,487 LOC. Of these changes, 414 were bug fixes involving 9,347 LOC. As we can see from the table, even though a Root only covers a small portion of the system, it is a hotspot where much maintenance effort was spent.

Table: Maintenance costs of each root
%: percentage; Rt. CF - BC: the total CF - BC of all files in each root

	Size (%)	Rt. CF (%)	Rt. CC (%)	Rt. BF (%)	Rt. BC (%)
root1	147 (10%)	1,109 (18%)	13,487 (21%)	414 (22%)	9,347 (25%)
root2	93 (6%)	1,050 (17%)	11,486 (18%)	452 (24%)	6,696 (18%)
root3	79 (5%)	601 (10%)	5,453 (9%)	183 (10%)	3,821 (10%)
root4	104 (7%)	486 (8%)	10,794 (17%)	166 (9%)	6,236 (17%)

(2) Extra maintenance costs of architecture roots: Considering each architecture root as a debt, DV8 provides a debt calculator to compute the penalty incurred by these roots. This penalty is calculated as the difference between the actual maintenance effort spent on the roots, and the expected maintenance effort spent on them. We use the average change/bug rate of all the files in a project as its expected maintenance effort [2]. The expected effort columns "*ExtraCF* "- "*ExtraBC*" represent the cumulative maintenance *penalty* from the roots. For example,"615" in the second row of "*ExtraBF* " column indicates that the 222 files in `root1` and `root2` are involved in bug fixes 615 times more often than average files. The "*Percentage*" row presents the percentage of the extra maintenance effort as compared with project averages. The last row indicates that, 28% of all the changes, 41% of all the LOC spent, 40% of bug-fixing changes, and 47% of bug-fixing LOC spent on the entire project are incurred by these roots.

Table: Extra maintenance costs of architecture roots.

	Penalty of Architecture Roots			
	Extra CF	Extra CC	Extra BF	Extra BC
root1	612	8,450	263	6,418
root2	1,332	16,601	615	11,175
root3	1,687	19,570	724	13,552
root4	1,754	26,110	763	17,314
Percentage	28%	41%	40%	47%

Please refer to the Quantify Architecture Debt section that explains how to use DV8 to quantify architecture debts, in the form of anti-patterns or roots.

[1] Ran Mo, Will Snipes, Yuanfang Cai, Srini Ramaswamy, Rick Kazman, Martin Naedele: **Experiences Applying Automated Architecture Analysis Tool Suites.** In *proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (ASE 2018). Pages 779-789

[2] Rick Kazman, Yuanfang Cai, Ran Mo, Qiong Feng, Lu Xiao, Serge Haziyev, Volodymyr Fedak, Andriy Shapochka: **A Case Study in Locating the Architectural Roots of Technical Debt.** In *Proceedings of the 37th International Conference on Software Engineering* (ICSE 2015). Page 179-188