Architecture Description Languages

Architecture Description Languages

- “ADLs are formal languages that can be used to represent the architecture of software intensive systems”

- Many languages:
  - Darwin, C2, Rapide, SADL, ACME, Wright, Unicon, ArchJava

- An ADL must explicitly model **components**, **connectors**, and their **configurations**

Using Types to Enforce Architectural Structure

Jonathan Aldrich (CMU)

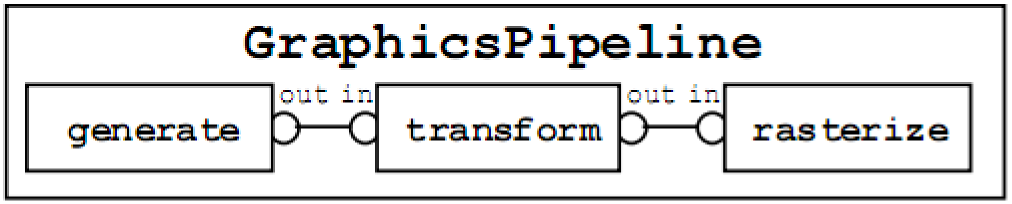
WICSA 2008

ArchJava

- ArchJava extends Java with:
  - Component classes: objects that are part of an architecture
  - Connections: allow components to communicate
  - Ports: endpoints of connections.

- Components are organized into a hierarchy using ownership domains

Example 1: Graphics Pipeline



- **generate** component: generates shapes to be displayed in the current scene
- These shapes are passed on to the **transform** component, which applies the current transformation to each shape in turn
- It then passes the shapes to the **rasterize** component to be drawn

Example 1: Graphics Pipeline

- We want to enforce two architectural invariants:
  - Components are arranged in a linear sequence, with each component getting information from its predecessor and sending it on to its successor
  - No data is shared between components

- These invariants support important quality attributes:
  - The ability to add and remove components from the pipeline
  - The ability to use a concurrent thread in each component

Graphics Pipeline in ArchJava

```
public component class GraphicsPipeline {
  protected Generate generate = ... ;
  protected Transform transform = ... ;
  protected Rasterize rasterize = ... ;

  connect pattern Generate.out, Transform.in;
  connect pattern Transform.out,Rasterize.in;

  public GraphicsPipeline() {             // construtora
    connect(generate.out, transform.in);
    connect(transform.out, rasterize.in);
  }
}
```

Graphics Pipeline in ArchJava

```
public component class Transform {
  protected Trans3D currentTransform;

  public port in {                           // provided
    methods provides void draw(unique Shape s);
  }

  public port out {                          // required
    methods requires void draw(unique Shape s);
  }

  void draw(Shape s) {
    currentTransform.apply(s);
    out.draw(s);
  }
}
```

Components

- A component is a special kind of object whose **communication patterns** are declared explicitly using architectural declarations
- **GraphicsPipeline** and **Transform** are component classes
- The **GraphicsPipeline** class contains three fields
  - One for each component in the pipeline
- **Generate** and **Rasterize** are component classes defined elsewhere
- **Trans3D** and **Shape** are ordinary classes that are not part of the architecture

Ports

- Components communicate through explicitly declared ports
- A port is a communication endpoint declared by a component.
  - The **transform** component class declares an **in** port that receives incoming shapes and an **out** port that passes transformed shapes on to the next component.
- Each port declares a set of required and provided methods
- A provided method is implemented by the component and is available to be called by other components connected to this port.
- Conversely, each required method is provided by some other component connected to this port.

Connect Patterns

- The declaration connect pattern **Generate.out, Transform.in** permits the graphics pipeline component to make connections between the **out** port of its **Generate** subcomponents and the **in** port of its **Transform** subcomponents
- Once connect patterns have been declared, concrete connections can be made between components
- The constructor for **GraphicsPipelin**e connects the out port of the **transform** component instance to the in port of the **rasterize** component instance.
- This connection binds the required method **draw** in the out port of **transform** to a provided method with the same name and signature in the in port of **rasterize**

Detecting Software Modularity Violations

Sunny Wong, Yuanfang Cai,
Miryng Kim, Michael Dalton

ICSE 2011

Abstract

- We present Clio, an approach that detects modularity violations, which can cause defects, modularity decay or expensive refactorings

- Clio computes the discrepancies between:
  - How components should change together based on the modular structure
  - How components actually change together as revealed in version histories

- We evaluated Clio using:
  - 15 releases of Hadoop Common
  - 10 releases of Eclipse JDT

Abstract

- The results show that hundreds of violations identified using Clio were indeed recognized as <u>design problems</u> or refactored by the developers in later versions

- The identified violations cover multiple symptoms of <u>poor design</u>, some of which are not easily detectable using existing approaches

Introduction

- Parnas' original definition of a module means an <u>independent task assignment</u>

- Information hiding principle advocates separating internal design decisions using an interface to allow for <u>independent evolution of other modules</u>

- The essence of software modularity is to allow for <u>independent module evolution and independent task assignment</u>

Introduction

- Rationale:
  - If two components always change together to accommodate modification requests
  - But they belong to two separate modules that are supposed to evolve independently
  - Then we consider this as a modularity violation

- First, we leverage Baldwin and Clark's design structure matrix (DSM) to determine structural coupling — how components should change together.

- Second, we mine the project's revision history to model change coupling — how components actually change together

Framework Overview

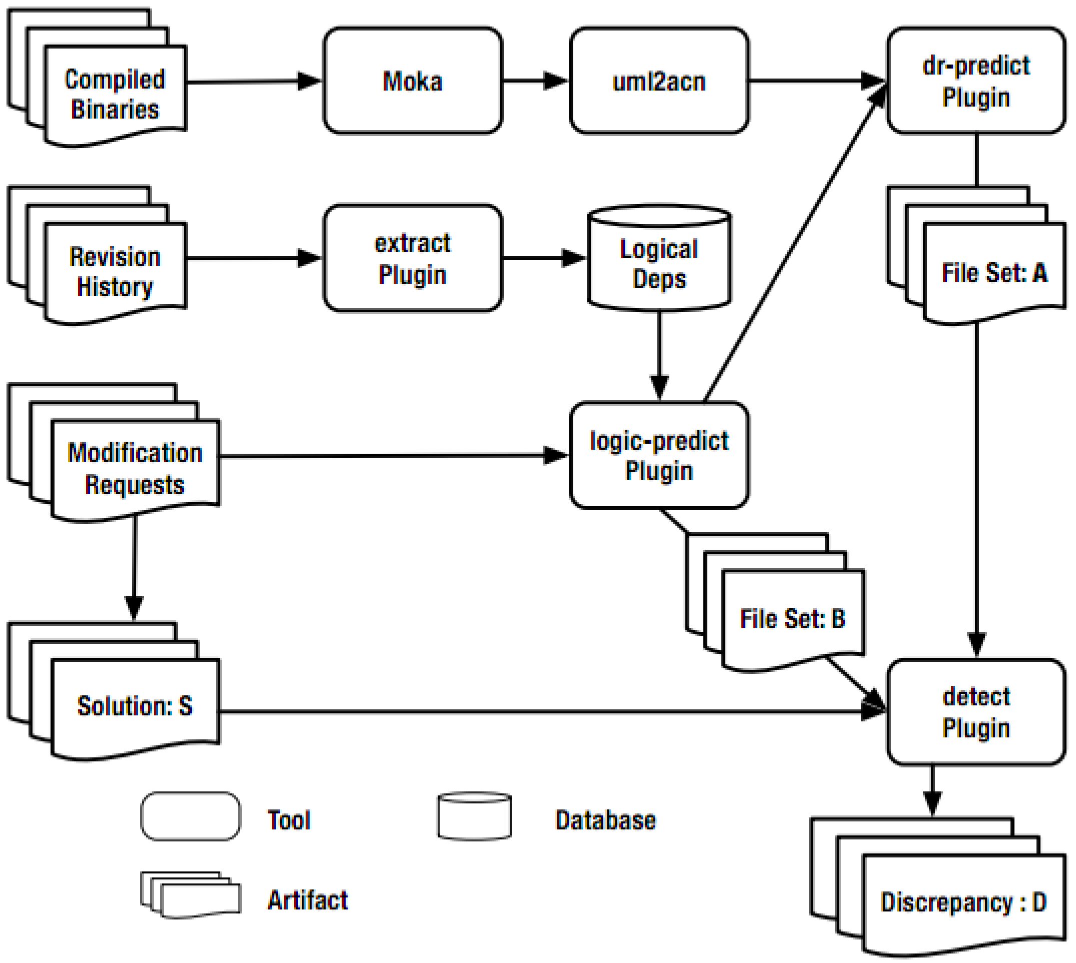
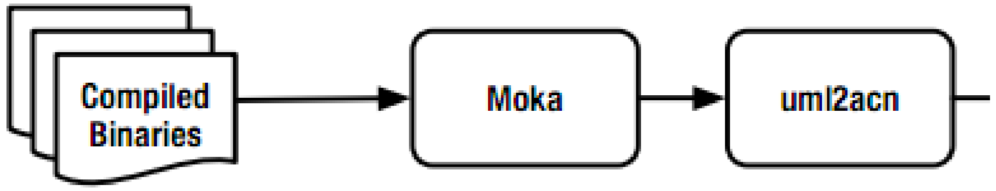


Figure 1: Approach Overview: the Clio Framework

DSM Extraction



- Clio uses the **Moka** tool to reverse engineer UML class diagrams from compiled Java binaries
- Clio then uses the **uml2acn** tool to transform a class diagram into DSM

Example: Maze Game

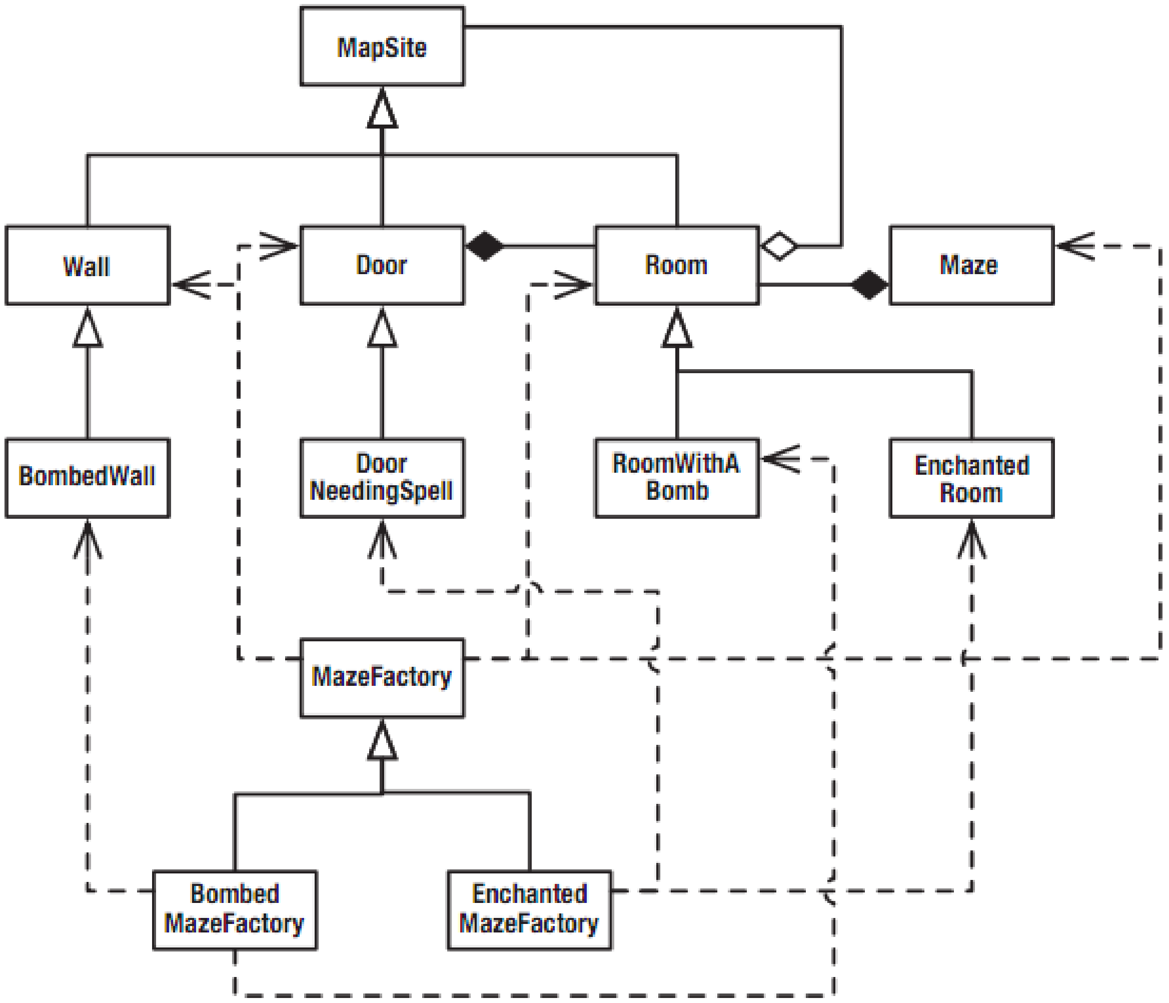
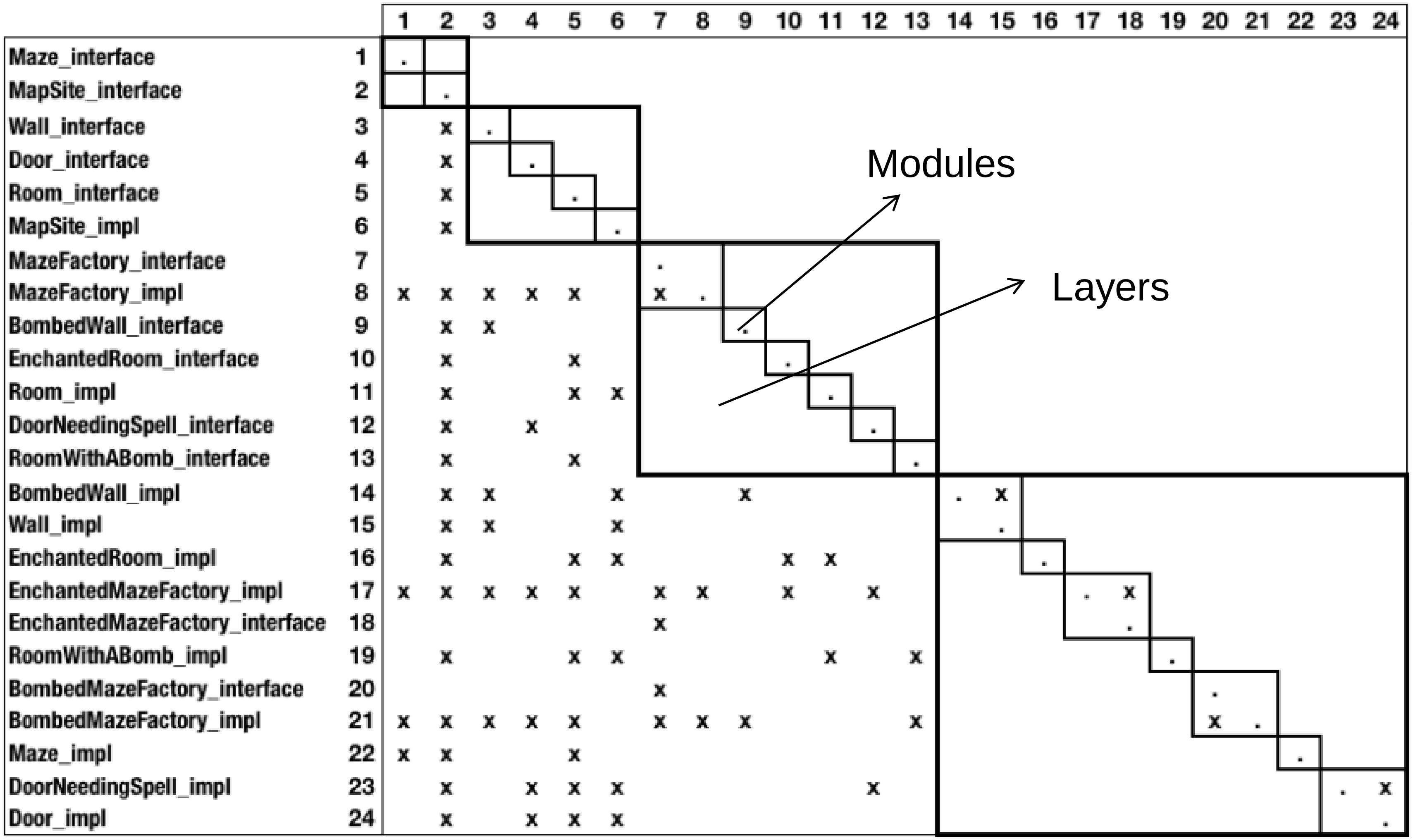


Figure 2: Maze game UML class diagram [28]

Maze Game DSM


Modules
Layers

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Maze_interface	1	.																							
MapSite_interface	2		.																						
Wall_interface	3		x	.																					
Door_interface	4		x		.																				
Room_interface	5		x			.																			
MapSite_impl	6		x				.																		
MazeFactory_interface	7							.																	
MazeFactory_impl	8	x	x	x	x	x		x	.																
BombedWall_interface	9		x	x						.															
EnchantedRoom_interface	10		x			x					.														
Room_impl	11		x			x	x					.													
DoorNeedingSpell_interface	12		x		x								.												
RoomWithABomb_interface	13		x			x								.											
BombedWall_impl	14		x	x			x			x					.	x									
Wall_impl	15		x	x			x									.									
EnchantedRoom_impl	16		x			x	x				x	x					.								
EnchantedMazeFactory_impl	17	x	x	x	x	x		x	x		x		x					.	x						
EnchantedMazeFactory_interface	18							x											.						
RoomWithABomb_impl	19		x			x	x					x		x						.					
BombedMazeFactory_interface	20							x													.				
BombedMazeFactory_impl	21	x	x	x	x	x		x	x	x				x							x	.			
Maze_impl	22	x	x			x																	.		
DoorNeedingSpell_impl	23		x		x	x	x						x											.	x
Door_impl	24		x		x	x	x																		.

Figure 4: Maze game DSM [28]

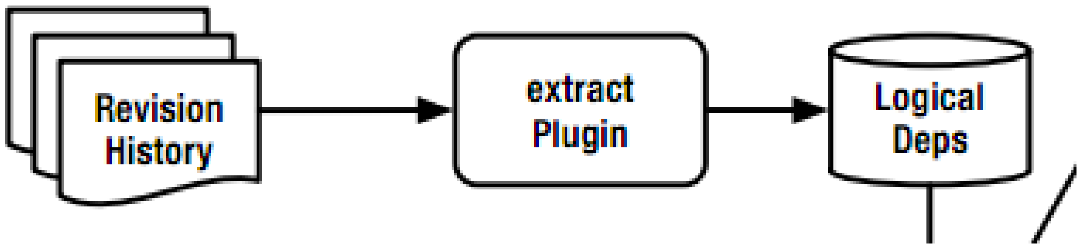
DSM Layers

- In order to identify modules, our prior work created a special clustering for DSMs
- Using this clustering, the columns and rows of the DSM can be reordered into layers
- The first layer in a DSM, L_1, is the group of variables clustered at the top left corner, and does not depend on any other layers
- A layer L_n only depends on layers L_{n-1} to L_1

[28] DSM Layers

- Layer 0 is the set of tasks that assume no other decisions.
- Layer i (i ≥ 1) is the set of all tasks that assume at least one decision in level i−1 and assume no decisions at a layer higher than i−1. Within any layer, no task assumes any decisions in another task of the same layer.
- Hence, the tasks within the same layer can be completed independently and in parallel.
- The highest layer is the set of independent modules. No decisions outside of these modules make assumption about any decisions within theses modules.

History-based Impact Analysis



- The second input is the revision history of the project, which is used to derive change couplings from a set of files changed to implement modification requests.

- The extract plugin of Clio computes change couplings at a file level, following the technique of Ying et al. [30]

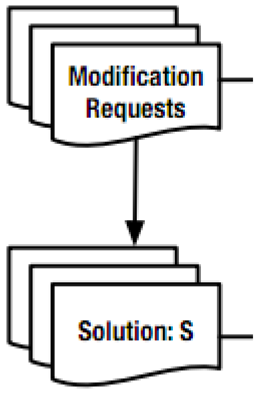
[30] Mining Change History

- We have been investigating an approach based on the mining of change patterns — <u>files that were changed together frequently in the past</u> — from a system's source code change history

- Mined change patterns can be used to recommend possibly relevant files as a developer performs a modification task.

- Our initial focus has been on the use of association rule mining to determine the change patterns. In this paper, we report on our use of an <u>association rule mining </u>algorithm: <u>frequent pattern mining</u>, which is based on frequency counts

Modification Requests



- The third input is the detailed information about a set of files S (called the MR solution), which was modified to <u>fulfill each modification request</u>

Clio Framework

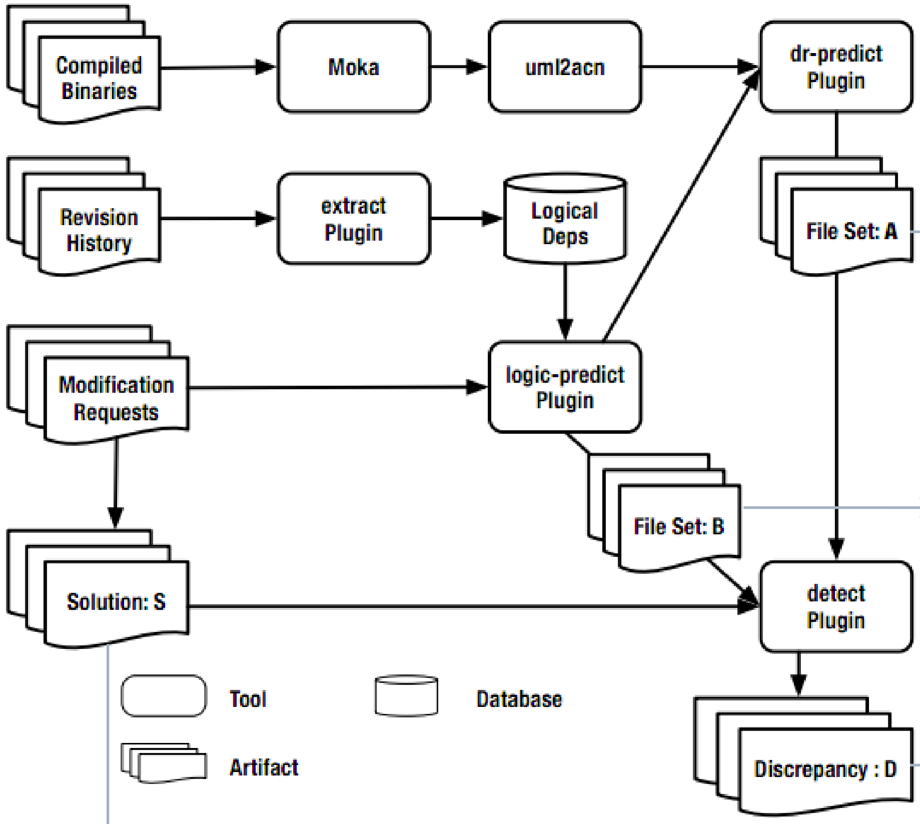


Figure 1: Approach Overview: the CLIO Framework

File Set A: components that are likely to be changed according to the original modular structure (for each modification request)

File Set B: components that are likely to be changed according to the co-changing patterns (for each modification request)

D= (B ∩ S) – A

By using B ∩ S, Clio filters out files that were accidentally changed together

Recurring discrepancies (a subset of D) are then reported to the users as violations

File set S (called the MR solution), components modified to fulfill each modification request

Example

- **EnchantedMazeFactory_impl** and **BombedMazeFactory_impl** are both located in the last layer of the DSM, meaning that they should evolve independently from each other.
- Clio's dr-predict plugin will never report that they are in each other's change scope
- If the revision history shows that they always change together, e.g. due to similar changes to cloned code, Clio will report that there is a modularity violation.

Evaluation

- Q1: How accurate are the modularity violations identified by Clio?
  - That is, do these identified violations indeed indicate problems?
  - We examine the project's version history to see whether and how many violations we identified in earlier versions are indeed refactored in later versions or recognized as design problems by the developers

- Q2: How early can Clio identify problematic violations?
  - For each confirmed violation, we compare the version where it was identified with where it was actually refactored or recognized by the developers

- Q3: What are the characteristics of design violations identified by our approach?

Subjects

- Two large-scale open source projects:
  - Hadoop Common
  - Eclipse Java Development Tools (JDT)

- Hadoop is a Java-based distributed computing system
  - We applied our approach to the first 15 releases
  - Covering about three years of development.

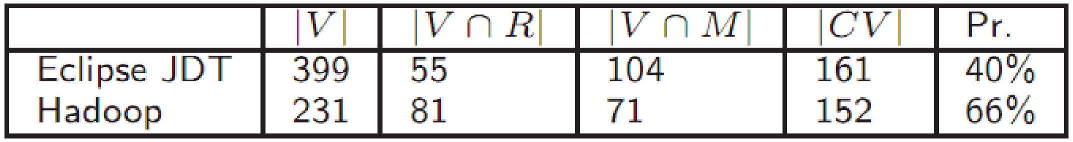
- Eclipse JDT is a core AST analysis took kit in the Eclipse IDE
  - We studied 10 releases of Eclipse JDT (2.0 to 3.0.2)
  - Also covering about three years of development

Evaluation Procedure

- We evaluate the output of Clio by checking the source code and MR records in later versions to see if they were indeed refactored or recognized as having a design problem
  - If so, we call such violation as being confirmed.
- We use both automated method and manual inspection to confirm a violation.

Results

Table 2: Modularity violations that occurred at least twice in the last five releases



	$\|V\|$	$\|V \cap R\|$	$\|V \cap M\|$	$\|CV\|$	Pr.
Eclipse JDT	399	55	104	161	40%
Hadoop	231	81	71	152	66%

- | V |: total number of violations reported by Clio
- | V ∩ R | : violations that match with automatically refactorings
- | V ∩ M |: remaining violations confirmed with manual inspection
- | CV |: total number of confirmed violations (|V ∩ R| + |V ∩ M|)

- Pr: precision (number of confirmed violations out of the total number of reported violations)

In-depth Case Study: Hadoop

- Automatically confirmed violations
- Manually confirmed violations
- False positive violations
- False negative violations

Manually Confirmed Violations

- Clio reported a violation in release 0.2.0 involving:
  - **TaskTracker**, **TaskInProgress**, **JobTracker**, **JobInProgress**, and **MapOutputFile**
  - Violation does not match with automatically reconstructed refactorings

- We searched Hadoop's MRs and found an open request MAPREDUCE-278, entitled "Proposal for redesign/refactoring of the JobTracker and TaskTracker".

- The MR states that these classes are "hard to maintain, brittle, and merits some rework." The MR also mentions that the poor design of these components have caused various defects in the system

Conclusions

- Parnas' original definition of a <u>module means an independent task assignment</u>
  - And his information hiding principle advocates separating internal design decisions using an interface to allow for independent evolution of other modules.

- Though this definition of modularity is inherently inseparable from the notion of independent module evolution, existing approaches do not detect modularity violations by comparing how components should change together and how the components actually change together