A LEARNING-BASED METHOD FOR DETECTING DEFECTIVE CLASSES IN OBJECT-ORIENTED SYSTEMS

Cagil Biray
Ericsson R&D Turkey

Assoc. Prof. Feza Buzluca
Istanbul Technical University

10th Testing: Academic and Industrial Conference
Practice and Research Techniques (TAIC PART)

Agenda

- INTRODUCTION
- HYPOTHESIS & OBSERVATIONS
- DEFECT DETECTION APPROACH
- CREATING THE DATASET
- CONSTRUCTING THE DETECTION MODEL
- EXPERIMENTAL RESULTS
- CONCLUSION
- Q&A

INTRODUCTION

SOFTWARE DESIGN QUALITY

- Definition:

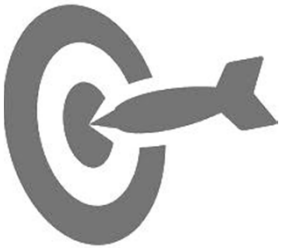
"capability of software product to satisfy stated and implied needs when used under specified conditions."

- How to assess the quality of software?
  - *Understandability, maintainability, modifiability, flexibility, testability...*
- Poorly designed classes include structural design defects.

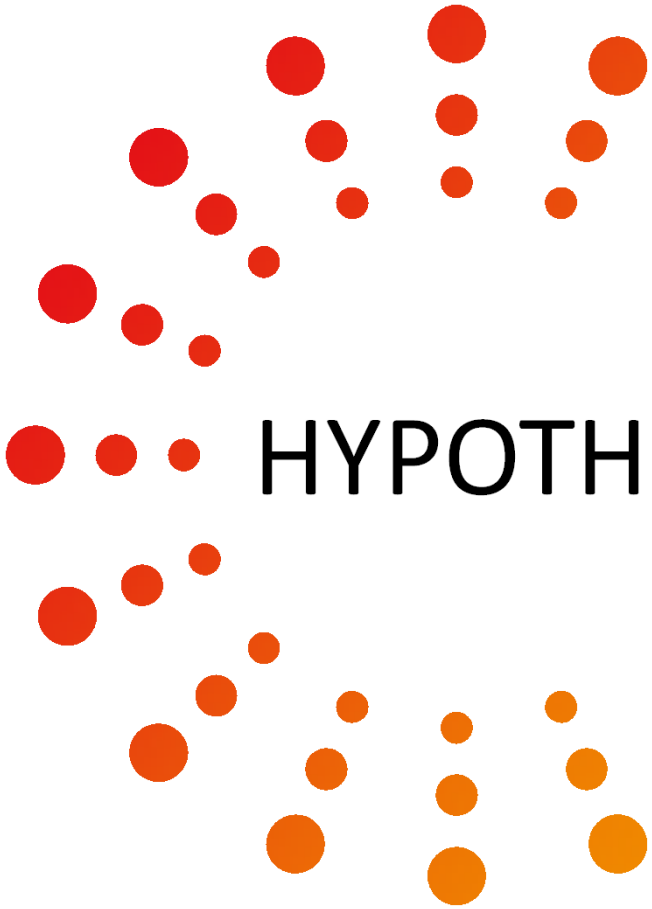
SOFTWARE DESIGN DEFECTS

- Structural defects are not detectable during compile-time or run-time.
- They reduce the quality of software as a cause the following problems:
  - ***Reduce the flexibility of software***
  - ***Vulnerable to introduction of new errors***
  - ***Reduce the reusability.***

OBJECTIVE



- Our **main objective** is to predict structurally defective classes of software.
- Two important benefits:
  - Helps testers to focus on faulty modules,
    - ✓ *Saves testing time.*
  - Developers can refactor classes to correct design defects,
    - ✓ *Reduces probability of errors.*
    - ✓ *Reduces the maintenance costs in future releases.*



HYPOTHESIS & OBSERVATIONS

HYPOTHESIS

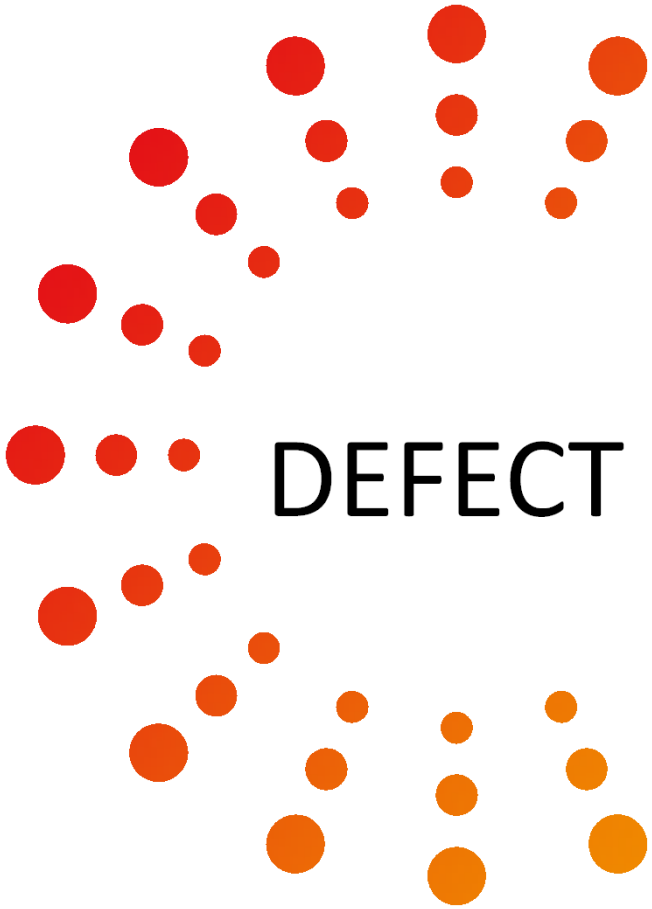
- Structurally defective classes mostly have following properties:
  - High class complexity, high coupling, low internal cohesion, inappropriate position in inheritance hiearchy.
- How to measure these properties?
  - Software design metrics

  Various metric types, distributions

  and different minimum/maximum values...

MAIN OBSERVATIONS

- Structurally defective classes tend to generate most of the errors in tests, but healthy classes are also involved in some bug reports.
- Defective classes may not generate errors if they are not changed; errors arise **after modifications.**
- Healthy classes are not changed frequently and if they are modified they generate errors very rarely.



DEFECT DETECTION APPROACH

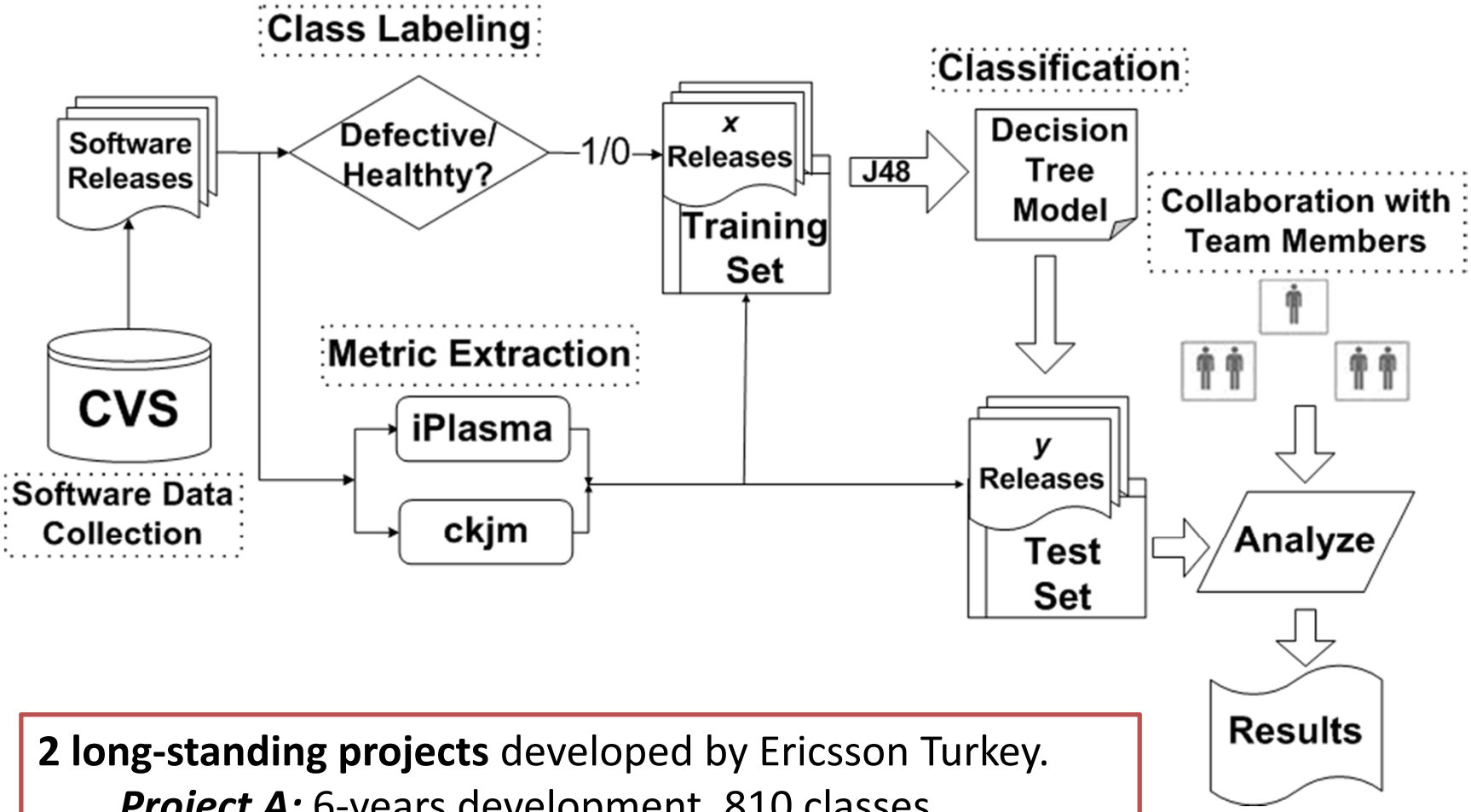
THE SOURCE PROJECTS

- 2 long-standing projects developed by Ericsson Turkey.
  - ***Project A:*** 6-years development, 810 classes.
  - ***Project B:*** 4-years development, 790 classes.
- **Release reports** of each project is analyzed.
  - Determine the reasons for changes
    - *Is it a **bug**?*
    - *Is it a **change request (CR)**?*

THE PROPOSED DEFECT DETECTION APPROACH

- A **learning-based method** for defect prediction: Learn from history, predict the future.
  - Rule-based methods, **machine-learning algorithms**, detection-strategies...
- How to construct **dataset**? (instances-attributes-labels)
  - Metric collection: iPlasma, ckjm tool.
  - Class labels: defective/healthy?
- How to create a learning **model**?
  - Decision trees.
    - J48 algorithm.

BASIC STEPS OF THE APPROACH



2 long-standing projects developed by Ericsson Turkey.

Project A: 6-years development, 810 classes.

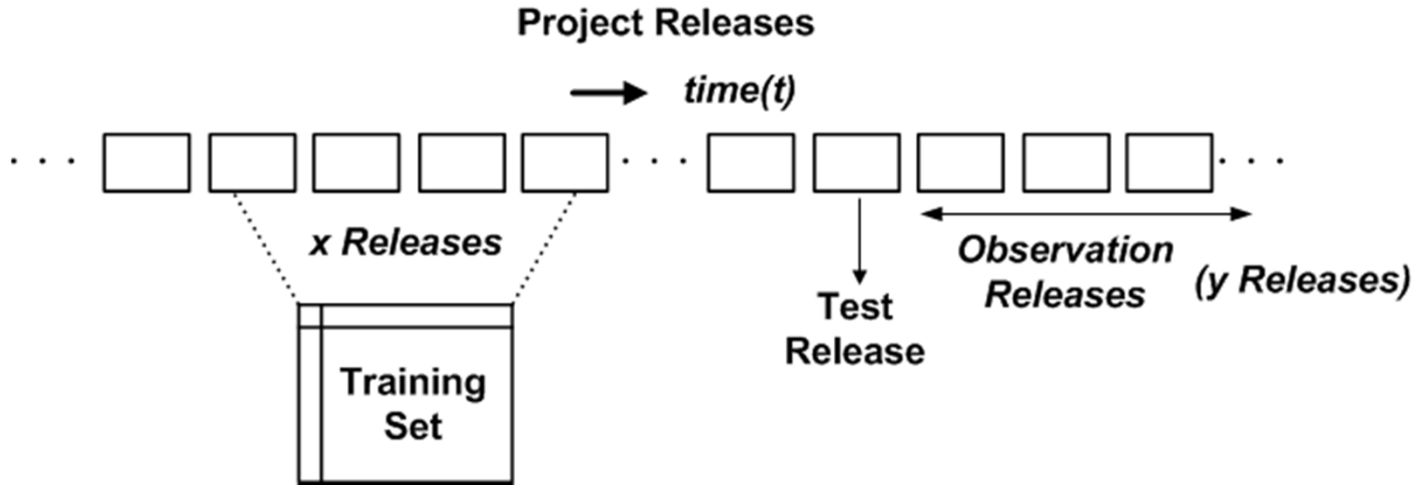
Project B: 4-years development, 790 classes.

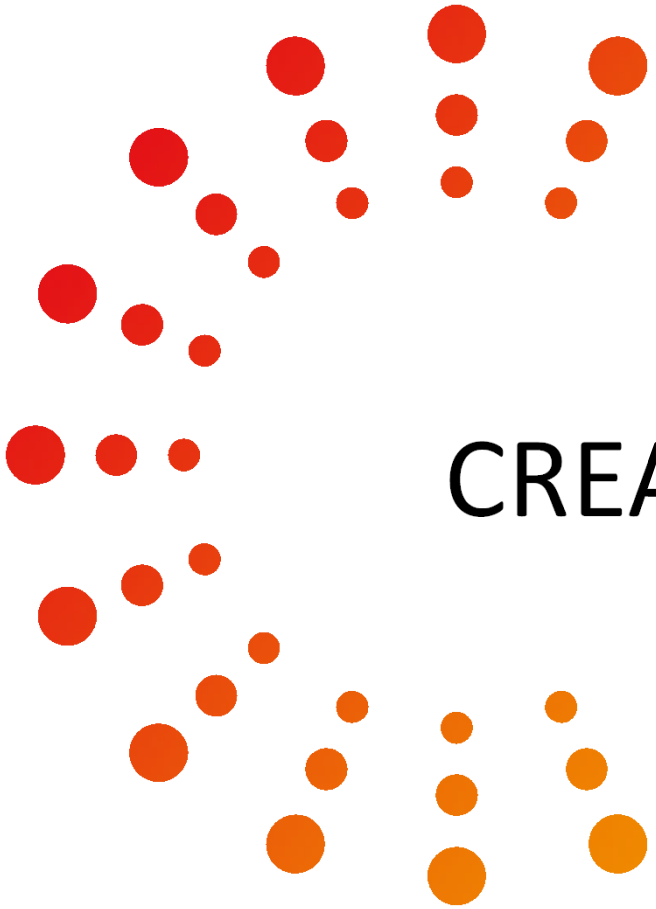
USING RELEASES FOR TRAINING AND EVALUATION

- We constructed the training set examing classes from 46 successive releases of the Project A.
- Applied model to test release of same project.
- Observed errors and changes in classes for 49 consecutive releases.
- Also, applied same model to a test release from Project B.
- Evaluated the performance of our method observing 49 releases of Project B.

USING RELEASES FOR TRAINING AND EVALUATION (cont' d)

- x = 46 consecutive releases (training set)
- y = 49 consecutive releases (observation releases)

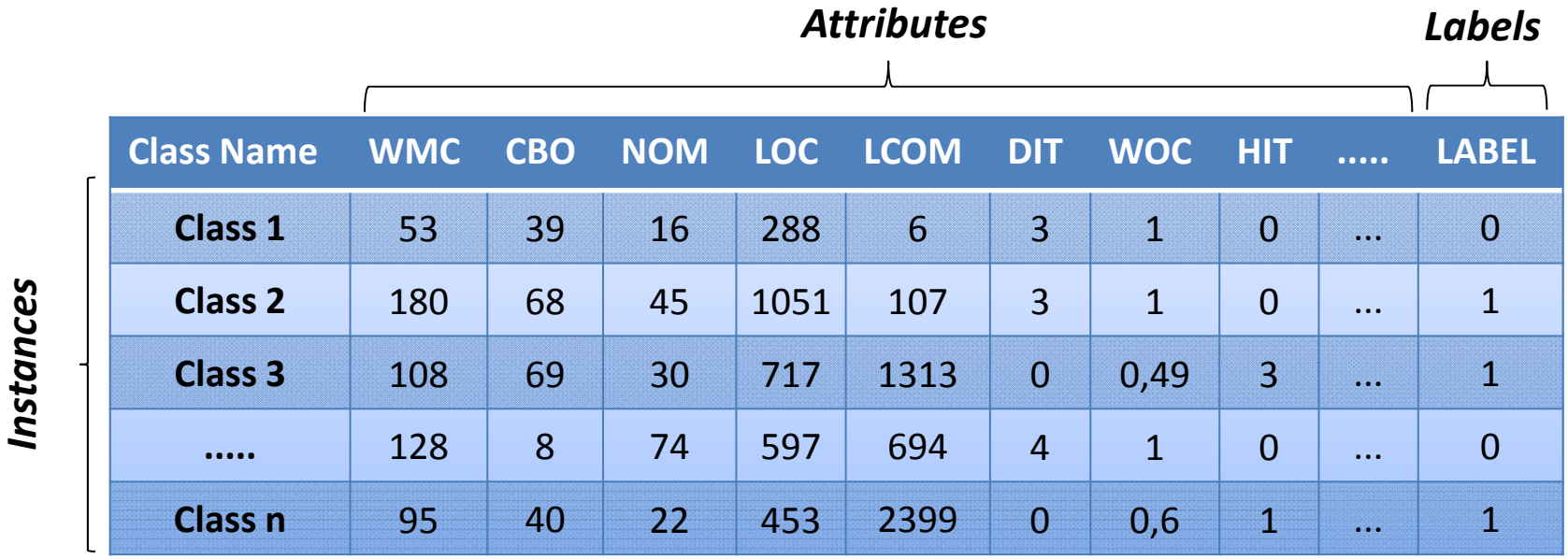




CREATING THE DATASET

CREATING THE DATASET

- Several releases of a project are examined to gather bug fix/CR information for each class.



Class Name	WMC	CBO	NOM	LOC	LCOM	DIT	WOC	HIT	.....	LABEL
Class 1	53	39	16	288	6	3	1	0	...	0
Class 2	180	68	45	1051	107	3	1	0	...	1
Class 3	108	69	30	717	1313	0	0,49	3	...	1
.....	128	8	74	597	694	4	1	0	...	0
Class n	95	40	22	453	2399	0	0,6	1	...	1

PARAMETERS of CLASS LABELING

- **ErrC (Error Count):** The total number of bug fixes which are made on a class in the observed *x* training releases.

$$ErrC_c = \sum_{i=1}^{x} e_{c,i}$$

- **CR (Change Request) Count:** The total number changes in the class made because of CRs of the customer.

$$CR\ count_c = \sum_{i=1}^{p} r_{c,i}$$

PARAMETERS of CLASS LABELING (cont' d)

- **ChC (Change Count):** The total number of changes in a class during the training releases.

$$ChC_C = ErrC_C + CR\ count_C$$

- **EF (Error Frequency):** The ratio between error count and change count of a class.

$$EF_C\ \% = \frac{ErrC_C}{ChC_C} * 100$$

THRESHOLD SELECTION

Training Set:

- Structural defective classes tend to change at **least 5** times and their EFs are **higher than 0.25**.

ChC ≥ 5
EF ≥ 0.25

✓ *t_1 is used for **ChC**, t_2 is used for **EF**.*

Error Frequencies		
Change Count	Error Count	Error Frequency
18	12	0.66
17	12	0.7
14	9	0.64
13	10	0.76
11	5	0.45
10	4	0.4
10	6	0.6
9	5	0.55
9	4	0.44
9	6	0.66
9	7	0.77
8	4	0.5
8	5	0.62
8	3	0.37
8	2	0.25
7	5	0.71
7	4	0.57
6	3	0.5
6	4	0.66
6	5	0.83

THRESHOLD SELECTION

- **Thresholds** are determined with the help of development team and experimental results.

- 2 thresholds for class labeling in training set:
  - *t_1 is used for ChC, t_2 is used for EF.*

  tag_c= Defective, if ($ChC_c \geq t_1$ and $EF_c \geq t_2$)

An Example: Defective Class

ChC > 3 & EF > 0.25

Release 4

BUG

Release Report

Class No.	Is a Bug?	Is a CR?	Error Count (ErrC)	CR Count	Change Count (ChC)	Error Freqency (EF)
1	YES	NO	1	0	1	1/1
1	NO	YES	1	1	2	1/2
1	YES	NO	2	1	3	2/3
1	YES	NO	3	1	4	3/4

An Example: Healthy Class

ChC > 3 & EF > 0.25

Release 4

CR

Release Report

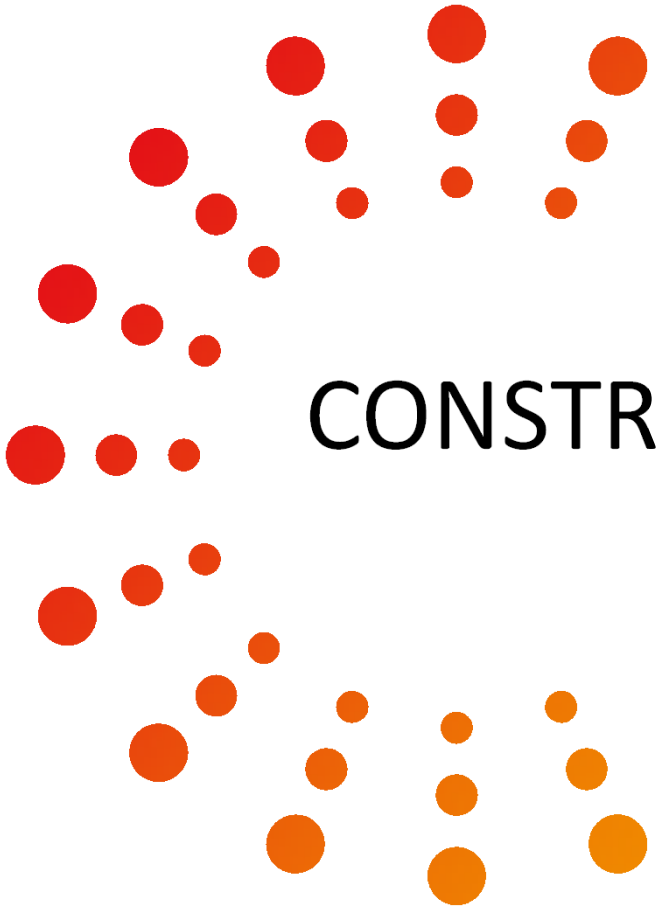
Class No.	Is a Bug?	Is a CR?	Error Count (ErrC)	CR Count	Change Count (ChC)	Error Freqency (EF)
1	YES	NO	1	0	1	1/1
1	NO	YES	1	1	2	1/2
1	NO	YES	1	2	3	1/3
1	NO	YES	1	3	4	1/4

What about 0/0 error frequencies?

RARELY & UNCHANGED CLASSES

- Not correct to tag them as "healthy".
- The common characteristic of high-EF classes: **complexity metric** (WMC) value is high.

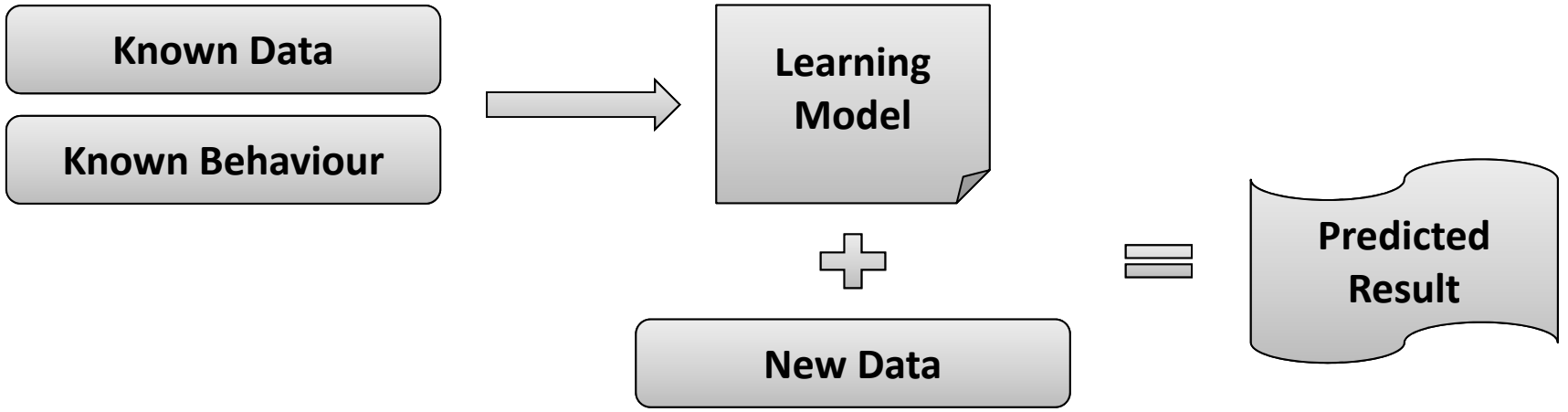
$$tag_c = \begin{cases} Defective,\ if\ (ChC_c \geq t_1\ and\ EF_c \geq t_2), \\ Defective,\ if\ ((ChC_c < t_1\ or\ EF_c < t_2)\ and\ WMC_c \geq AVG*1.5), \\ Healthy,\ otherwise. \end{cases}$$



CONSTRUCTING THE DETECTION MODEL

CONSTRUCTING THE DETECTION MODEL

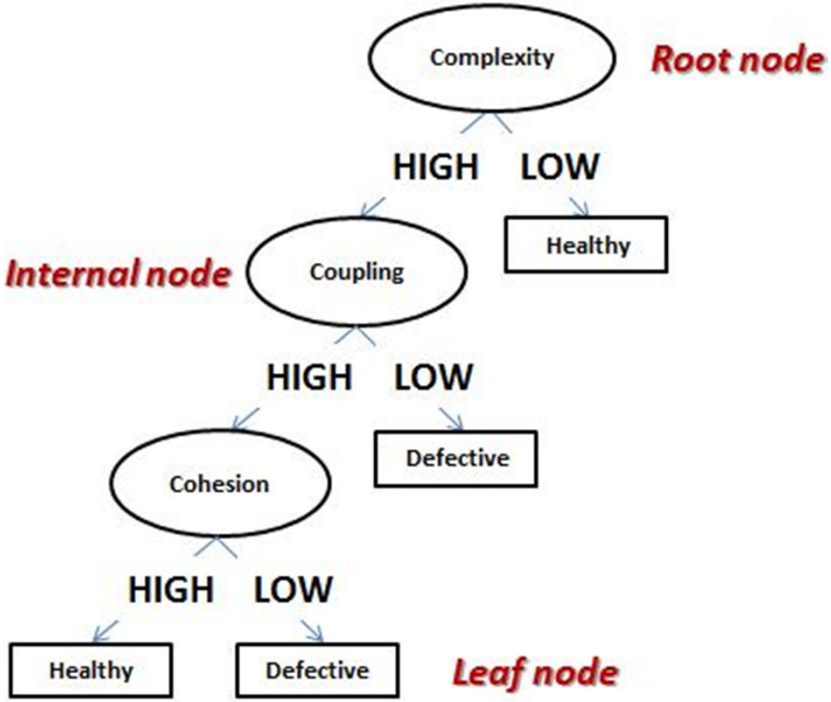
- A classification problem within the concept of machine learning.



- J48 decision-tree learner.

DECISION TREE ANALYSIS

- J48 algorithm selects metrics strongly related to defect-proneness of the classes.



EXPERIMENTAL RESULTS

CREATING THE TRAINING SET

Expression	Quantity	Classification Label
ChC ≥ 5 and EF ≥ 0. 25	45	Defective
(ChC < 5 or EF < 0.25) and $WMC_c \geq AVG(WMC_{dc})*1.5$	2	Defective
(ChC < 5 or EF < 0.25) and $WMC_c < AVG(WMC_{dc})*1.5$	200	Healthy

- **247** classes, **23** object-oriented metrics and **defective/healthy class tags** in data set.
- **J48 classifier** algorithm selected 5 metrics: *CBO, LCOM, WOC, HIT and NOM*.

RESULTS OF EXPERIMENTS (Project A)

- We applied unseen test release to decision tree model.
- Predictions
  - 53 out of 807: defective
  - 81% of the most defective classes
  - 18 classes with 0/0 EFs: 13 of them are defective.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
8 / 11 = 0.73	1	1
7 / 11 = 0.64	1	0
6 / 12 = 0.5	1	1
6 / 10 = 0.6	1	1
6 / 7 = 0.86	1	1
5 / 11 = 0.45	1	1
5 / 10 = 0.5	1	1
5 / 9 = 0.56	1	0
5 / 8 = 0.63	1	1
5 / 7 = 0.71	1	1
4 / 10 = 0.4	1	1
4 / 6 = 0.67	2	0
4 / 5 = 0.8	1	1
3 / 7 = 0.43	2	2
3 / 6 = 0.5	1	1
3 / 5 = 0.6	2	2
2 / 5 = 0.4	2	2

RESULTS OF EXPERIMENTS (Project B)

- Predictions
  - 41 out of 789: defective
  - 83% of the most defective classes.
  - 7 classes with 0/0 EFs: 4 of them are defective.

ErrC / ChC = EF	Total # of Defective Classes	Total # of Correctly Detected Classes
10 / 10 = 1	1	1
9 / 11 = 0.82	1	1
8 / 9 = 0.89	1	1
7 / 7 = 1	1	1
6 / 8 = 0.75	1	1
6 / 7 = 0.86	1	0
5 / 6 = 0.83	1	1
5 / 5 = 1	1	1
4 / 5 = 0.8	2	2
3 / 6 = 0.5	1	0
3 / 5 = 0.6	1	1

CONCLUSION

CONCLUSION

- Our proposed approach ensures the early detection of defect-prone classes and provides benefits to the developers and testers.
- Helps *testers* to **focus on faulty modules** of software: saves significant proportion of testing time.
- *Developers* can **refactor** classes to correct their design defects: reduce the maintenance cost in further releases.

Q&A

Thank you.