介绍

1. 当前DPD存在的问题

- 设计模式存在变体（varients）问题：
  - 一种抽象的解决方案可能有不同的实现，虽然这些实现有相同行为
  - 模式的定义不够正式，不同的开发者有不同的理解，导致实现有结构不同但行为相同的变体
- 由于变体问题的存在，对设计模式的理解和探测都是主观的
- 上述问题导致给一种设计模式定义完备、封闭的规格说明（specification）或者探测规则（detection rule）是不可能的

2. 解决方案

- 使用机器学习的方法可以解决上述问题
  - 可以通过改变样本集（example set）来改变探测规则（detection rule），使探测规则聚焦于不同的变体
- 使用了两种模块Joiner和Classifier，对于某种给定的需要探测的设计模式：
  - Joiner负责初步筛选该模式的候选者（Joiner具有高recall和低percision）
  - Classifier负责判断Joiner选出的候选者是否正确，并给出该判断的置信度（这一步使用了机器学习）
  - 这么做的好处是：classifier计算量小；classifier对训练集的数量要求相对较小；classifier不依赖具体的机器学习技术（可以用任何分类方法）

主要贡献

- 对设计模式的建模，该模型可以把设计模式表示成元素和元素间的关系
- 把DPD（设计模式探测）定义成了一种有监督的分类任务
- 手工标注了一个关于设计模式的大数据集

项目背景

1.code entities

- 代码实体是可以用名称（name）唯一标识的代码结构
- 在OO中指类（包含接口、枚举类型、注解类型）、方法、成员变量

2. micore structures

- 可以看成一对代码实体之间的关系，有一个source和一个destination
- source和destination可以是同一个代码实体
- 作者使用了三种不同的微结构（elemental design patterns 、design patterns clues和micro patterns ），每种都能包括多个微结构
- 尽管三种微结构有区别，但是它们都是设计模式结构的组成部分，也都可以拿来探测设计模式，因此后面对他们的处理上没有区别

3. design pattern detection

- 设计模式的定义描述了组成该模式的角色（role）
- “role mapping”：把类分配给角色；在一个role mapping中，一个角色只能对应一个类，但一个类可以对应多个角色

进行DPD的工具MARPLE

MARPLE是Metrics and ARchitecture Reconstruction PLugin for Eclipse

1. 架构图

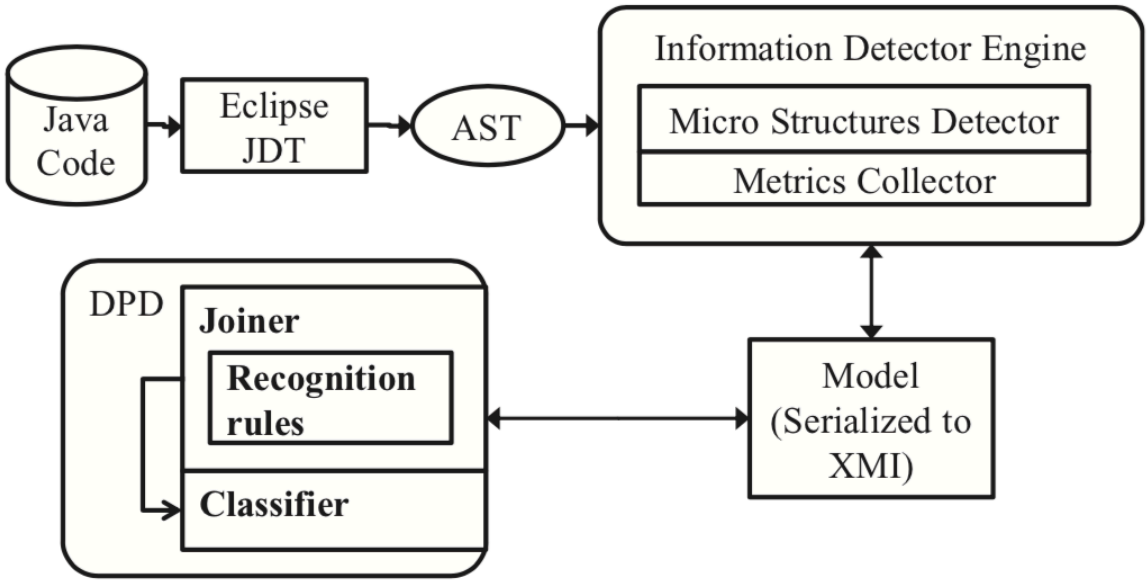


Figure 1. The architecture of MARPLE.

2. 工作流程

- 首先使用JDT插件将源码构建成抽象语法树（AST）
- Information detector engine：建立整个系统的模型、收集micore structures和metrics（系统的一些度量信息）并把它们存在模型中
- Joiner从微结构中找出所有满足给定定义的设计模式实例的候选者
- Classifier判断Joiner找出的候选者是否正确

3. 对重要模块的介绍

- Micro-structures detector（MSD）
  - 一组访问者解析源代码的AST，每个访问者支持检测一个微结构
  - 一个微结构和一段(或一组)代码之间总是存在1对1的对应关系，所以该检测具有100%的准确率和召回率
- Joiner
  - 分析来自MSD的信息，以从系统中找出类组（groups of classes），其中每个组都是模式候选者
  - 在模式候选者中，将角色分配给每个类，并根据树结构组织，树结构对模式的概念组织进行建模
  - Joiner把系统看成图，微结构是边，代码实体（TYPE？？？）是节点，使用图匹配技术提取设计模式的候选者

- 具体技术：图描述使用RDF、图匹配使用SPARQL查询
- Classifier
  - 评估模式实例（候选者）与预先分类的设计模式的相似性，以决定评估的实例是否正确
  - 每次评估完新实例后，都可以将它们添加到存储库中
  - 存储库中的模式都可以用于再次训练机器学习模型，模型更新后可以继续用于新的分类任务

探测（detection）流程

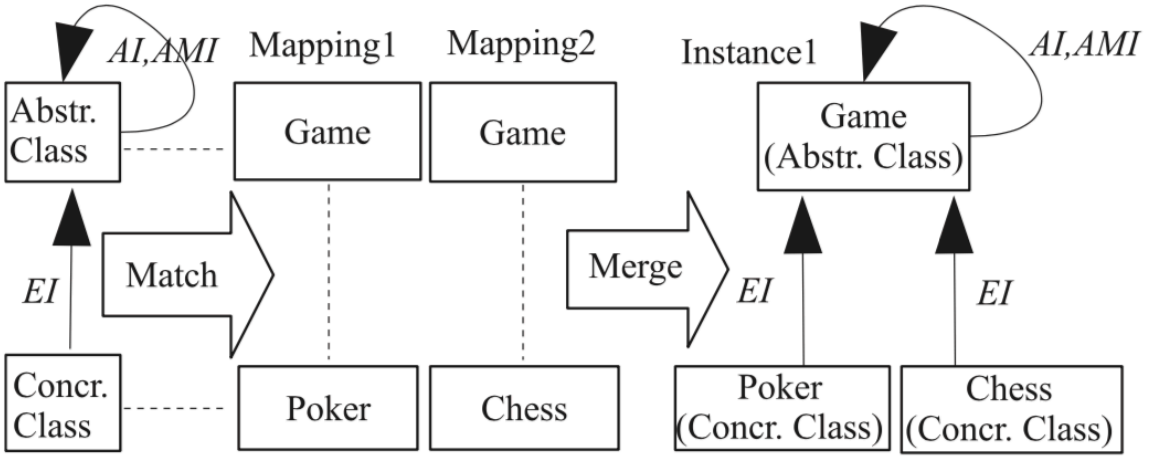


Figure 2. Joiner detection process example for the Template Method DP. Micro-structures: EI = extended inheritance, AI = abstract interface, AMI = abstract method invoked.

1. 提取设计模式的候选者

- 输入是再系统上检测到的微观结构列表，以图的形式表示
- 分为两个过程：Maching和Merging
- Maching：输出是**一组**角色映射（**set of** role mapping）
  - 每个角色映射有固定的长度，等于在设计模式定义中声明的角色数量
  - 同一个类可以映射到不同的角色
  - $$rm : Roles \rightarrow Classes \quad (1)$$
  - $$Matched = \{rm_1, \ldots, rm_i, \ldots, rm_n\} \quad (2)$$
  - 以图二为例，**rm1 = [Abstr. Class -> Game, Concr. Class -> Poker]，rm2 = [Abstr. Class -> Game, Concr. Class -> Chess]，Mached = {rm1, rm2}**
- Merging：同一个设计模式实例可能包含了不同的rm，这些rm被分散在了Mached集合中。Merging步骤的目的就是将这些属于同一个实例的rm合并在一起
  -

$$
\begin{aligned}
DpInstance &::= instance : LevelInstance, definition : \underline{DpDef} \\
DpDef &::= level : LevelDef, name : String \\
LevelDef &::= roleDefs : RoleDef^{+}, children : LevelDef^{*} \\
RoleDef &::= name : String \\
LevelInstance &::= levels : Level^{*}, roles : RoleAssociation^{+} \\
Level &::= instances : LevelInstance^{+}, definition : \underline{LevelDef} \\
RoleAssociation &::= role : \underline{RoleDef}, className : String
\end{aligned}
$$

- 首先使用GEBNF定义了层级（level），层级用于描述设计模式中各个角色的层次关系。一个设计模式可以包含多个level，level将role组织成树的结构
- 算法：目标是将一组角色映射填充到给定的结构中
- merge() 函数的输入是Mached集合以及给定的dp根级定义，输出dp的所有实例集
-

Algorithm 1 merge(m: *Matched*, dp: *DpDef*):

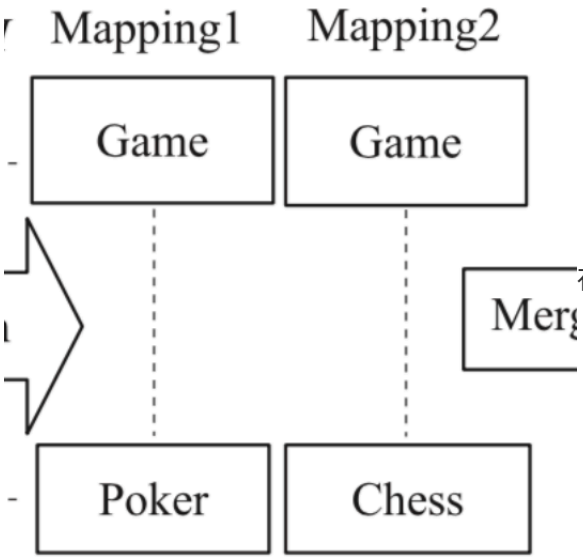
```
l ← newLevel(dp.level)
for all rm ∈ m do
    addLevel(rm, l) // add every mapping to the proper sublevel in l
end for
merge ← l.instances
```

-

Algorithm 2 addLevel(rm: RoleMapping, level: Level):

```
if hasInstance(level, rm) then // get level instance if existing
    li ← getInstance(level, rm)
else // otherwise create a new one
    li ← newLevelInstance(level)
    for all r ∈ level.definition.roles do // add its role mappings
        li.roles ← li.roles ⋃ newRoleAssociation(r, rm(r))
    end for
    for all l ∈ level.definition.children do // and create its sublevels
        li.levels ← ⋃ newLevel(l)
    end for
end if
for all sl ∈ li.levels do // add instances and mappings sublevels
    addLevel(rm, sl)
end for
```

- 具体算法：
  - 给定Mached集合和一个根级定义DpDef，每个角色映射都添加为root level的一个child

- 对于给定rm和当前Level，将检索或创建一个LevelInstance li（调用getInstance）；
- 什么时候创建什么时候不创建： 

在遇到Mapping1中的absractCalss - Game这一映射时，因为是第一次遇到，所以创建；在遇到Mapping2的abstractCalss - Game这一映射时，因为之前已经遇到过了，所以直接返回之前创建的实例
  - 这一步其实就做到了Merge
- 根据当前level的levelDef中的roleDefs中定义的role，将rm中相应的role分配给levelInstance（li）的roles属性
- 对当前level的每一个子level，递归调用上述过程，直至没有子level，此时rm中的role被添加完毕
- 对Mached集合中的其他rm，重复上述过程

- 单级模式：所有角色都在同一个层级上的模式，包括单例和适配器
- 多级模式：其他的模式

2. 分类过程

- 输入：
  - 模式定义：需要提取的设计模式的模式定义
  - 微结构：在分析项目代码过程中收集的所有微结构的集合。需要注意的是，在选择候选者阶段已经使用了一小部分微结构，在分类阶段将使用那些没有被Joiner使用的微结构进行分类
  - 模式存储库：已分类的设计模式实例的存储库
  - 模式候选者：由Joiner模块找出的候选者
- 步骤：
  - 从模式实例到角色映射
  - 从角色映射到模式实例

用于分类的模式实例建模

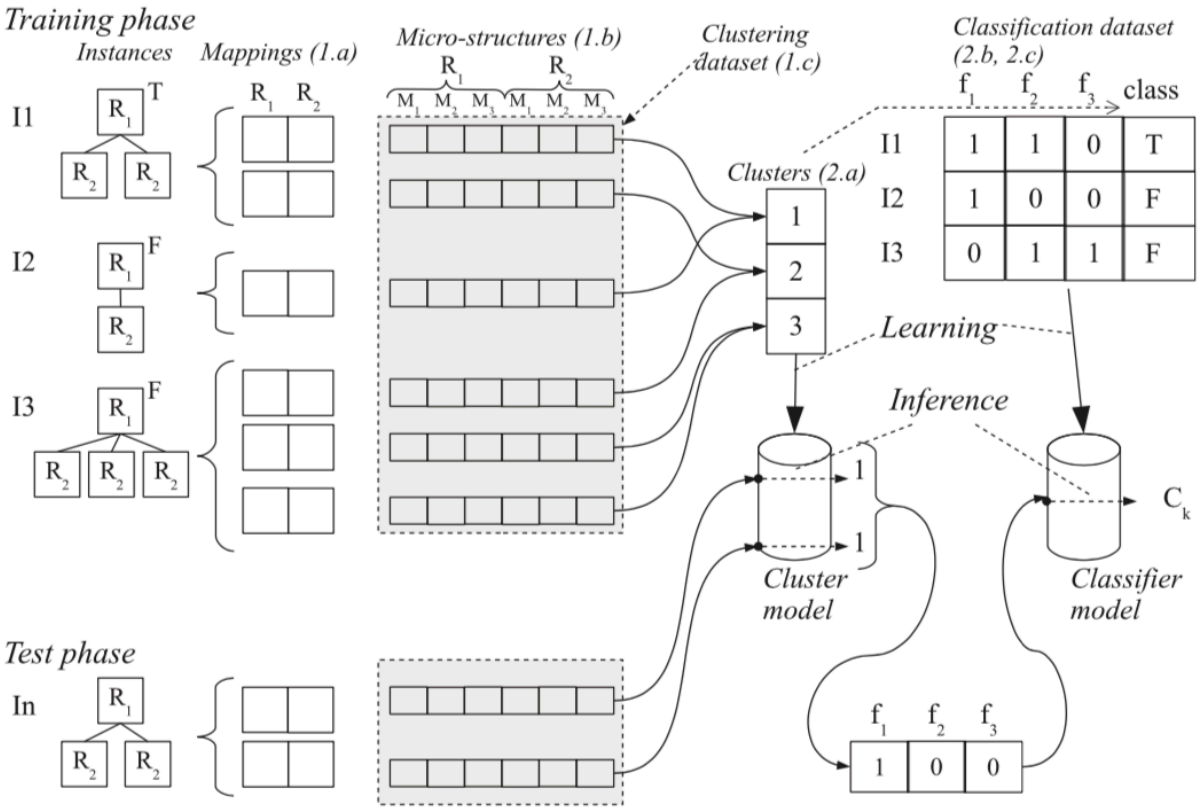


Figure 3. Complete classification process (simplified: the complete representation of the clustering dataset is shown in Table 2 for space reasons).

1. 从模式实例到角色映射

- 使用用于创建模式实例的角色映射列表来表示模式实例
- 将上述表示（角色映射列表）和类的微结构信息组合在一起
- 形成一种新的表示形式
- 举例：

Table 2
Clustering dataset example.

Map#	$R_1M_1R_1$	$R_1M_1R_2$	$R_1M_2R_1$	$R_1M_2R_2$	$R_1M_3R_1$	$R_1M_3R_2$	$R_2M_1R_1$	$R_2M_1R_2$	$R_2M_2R_1$	$R_2M_2R_2$	$R_2M_3R_1$	$R_2M_3R_2$
1	1	1	1	0	0	1	1	0	1	1	0	1
2	0	1	1	0	0	0	1	0	1	0	1	1

  - 假设一个实例候选者由两个role mapping组成，每个role mapping包含两个role（R1和R2）
  - 未被Joiner使用的微结构有三种：M1，M2，M3
  - 微结构表示代码实体之间的关系，而在一个role mapping中，每个role也被分配了一个代码实体，因此，可以用微结构表示两个role之间的关系
  - 把他们组合到一起，可以得到12种不同的特征，每个映射都用这些特征表示，其中，RiMjRk表示角色Ri和Rk之间具有Mj关系
  - 得到的所有映射组成的数据集被称为clustering dataset
- 这个clustering dataset不能直接拿去分类
  - 原因：每个模式实例具有的rm的数量是不确定的，因此得到的特征向量的数量也是不确定的
  - 解决方案：使用聚类方法对特征向量再进行一次编码，即步骤二

2. 从角色映射到模式实例

- 过程：

    - 将上一个步骤得到的clustering dataset拿来训练聚类算法（软聚类）
    - 聚类算法和类型的数量k本过程不做要求，可由开发者自行指定
    - clustering dataset中的映射被分配到不同的簇（cluster）中
    - 得到实例的新的特征：对于每一个实例，如果这个实例的所有映射中至少有一个映射被分配到了第i个簇，那么新特征的第i项是1，否则是0
    - 举例：图3中实例I1有两个role mapping，在上一步得到了两个映射。聚类算法得到了三个簇，其中I1的第一个映射被分配到了第一个簇中，第二个映射被分配到了第二个簇中，第三个簇中没有I1的映射，所以I1的特征为（1，1，0）
    - 得到的数据集被称为classification dataset，在该数据集中每个实例唯一对应一条向量，且各向量的长度相同
    - 使用classification dataset进行分类
- 训练阶段：
    - 对使用训练集得到的clustering dataset进行聚类操作时，得到一个训练好的聚类模型
    - 人工对训练集的classification dataset打上标签（True或Flase），训练分类器
- 测试阶段：
    - 使用训练好的聚类模型进行聚类操作
    - 使用训练好的分类器进行分类
- 该过程解决了模式实例大小未知的问题
- 值得注意的是，在单层设计模式中，所有角色都在表示模式的树的根中定义。这导致每个实例只包含一个role mapping。本来使用聚类的目的就是为了解决role mapping数量不知道的问题，因此这里用了一种自定的聚类算法：如果映射的第i个特征的值为真，则簇i包含这个映射。这样，得到的特征向量实际上和表示映射的向量是一样的。
    - 实际上就是没有使用聚类，这个算法只是为了模型的统一

实验部分

- 目标：
    - 测试机器学习模型相对于基线模型的优越性
    - 选择最有效的机器学习模型和聚类算法
- 使用了10倍交叉验证
- 数据集：选择了10个项目

Table 3
Projects for the experimentations.

Project	CUs	Pack	Types	Meth	Attr	TLOC
DPExample	1060	235	1749	4710	1786	32, 313
QuickUML 2001	156	11	230	1082	421	9233
Lexi v0.1.1 alpha	24	6	100	677	229	7101
JRefactory v2.6.24	569	49	578	4883	902	79, 732
Netbeans v1.0.x	2444	184	6278	28,568	7611	317,542
JUnit v3.7	78	10	104	648	138	4956
JHotDraw v5.1	155	11	174	1316	331	8876
MapperXML v1.9.7	217	25	257	2120	691	14,928
Nutch v0.4	165	19	335	1854	1309	23,579
PMD v1.8	446	35	519	3665	1463	41,554

CUs: Number of compilation units, TLOC: Tot number of lines of code.
Pack: Number of packages.

- 使用Joiner探测到的模式实例，将评估的数量限制在1000个左右是为了权衡数据量和手工评估的工作量

Table 4
Summary of detected pattern instances.

Pattern	Candidates	Evaluated	Correct	Incorrect
Singleton	154	154	58	96
Adapter	6733	1221	618	603
Composite	128	128	30	98
Decorator	250	247	93	154
Factory method	2875	1044	562	482

- 算法选择和参数优化
  - 使用的算法：ZeroR, OneR, Naïve Bayes, JRip, Random Forest, C4.5, SVMs (with different kernel functions), SimpleKMeans, CLOPE.
  - 选择的算法包含了只吃分类的主要算法，即 probabilistic, separation 和 heuristic approaches
  - 使用accuracy, F1-measure 和 area under ROC三个指标来表示算法的性能，也是根据这三个指标进行调参
- 结果

Table 6
Best performances on single-level design patterns.

Classifier	Singleton			Adapter		
	Acc.	F_1	AUC	Acc.	F_1	AUC
ZeroR	0.61	0.00	0.48	0.53	0.00	0.50
OneR	0.87	0.83	0.86	0.68	0.66	0.68
NaiveBayes	0.81	0.73	0.89	0.70	0.70	0.78
JRip	0.88	0.85	0.88	0.81	0.77	0.82
RandomForest	**0.93**	0.90	**0.97**	0.85	0.84	0.92
J48 Unpruned	0.87	0.82	0.91	0.80	0.79	0.86
J48 Reduced Error Pruning	0.88	**0.91**	0.85	0.79	0.79	0.84
J48 Pruned	0.88	0.84	0.91	0.80	0.79	0.86
SMO RBF	0.90	0.86	0.94	0.85	0.84	0.92
C-SVC Linear	0.83	0.77	0.87	0.80	0.79	0.85
C-SVC Polynomial	0.91	0.88	0.94	0.84	0.82	0.89
C-SVC RBF	0.92	0.89	0.95	**0.86**	**0.85**	0.92
C-SVC Sigmoid	0.86	0.88	0.94	0.79	0.69	0.84
ν-SVC Linear	0.91	0.88	0.93	0.80	0.79	0.85
ν-SVC Polynomial	0.90	0.87	0.94	0.84	0.82	0.91
ν-SVC RBF	0.91	0.89	0.95	**0.86**	0.84	**0.93**
ν-SVC Sigmoid	0.90	0.79	0.92	0.75	0.64	0.82

Acc: Accuracy, F_1 : F1-measure, AUC: area under ROC.

Table 7
Best performance results for the multi-level patterns.

Classifier	Composite			Decorator			Factory Method		
	Acc.	F_1	AUC	Acc.	F_1	AUC	Acc.	F_1	AUC
Clusterer: SimpleKMeans									
ZeroR	0.75	0.00	0.39	0.58	0.00	0.49	0.52	0.69	0.50
OneR	0.75	0.11	0.50	0.70	0.56	0.65	0.70	0.73	0.70
NaiveBayes	0.77	0.38	0.63	0.77	0.74	0.76	0.81	0.82	**0.87**
JRip	0.75	0.35	0.48	0.80	0.76	0.76	0.76	0.79	0.77
RandomForest	0.75	0.45	0.61	**0.82**	**0.77**	**0.82**	**0.82**	**0.83**	**0.87**
J48 Unpruned	0.75	0.27	0.59	0.77	0.75	0.74	0.80	0.81	0.86
J48 Reduced Error Pruning	0.77	0.25	0.51	0.77	0.73	0.77	0.78	0.81	0.85
J48 Pruned	0.75	0.00	0.51	0.80	0.76	0.75	0.79	0.81	0.85
C-SVC RBF	0.79	0.36	**0.88**	0.80	0.75	**0.82**	**0.82**	**0.83**	0.84
ν-SVC RBF	**0.81**	0.55	0.67	0.80	0.76	0.81	0.80	0.82	**0.87**
Clusterer: CLOPE									
ZeroR	0.75	0.00	0.39	0.58	0.00	0.49	0.52	0.69	0.50
OneR	0.75	0.00	0.50	0.66	0.59	0.64	0.54	0.69	0.53
NaiveBayes	0.75	0.12	0.52	0.70	0.67	0.73	0.55	0.69	0.55
JRip	**0.81**	0.42	0.60	0.71	0.65	0.68	0.54	0.69	0.52
RandomForest	**0.81**	0.38	0.65	0.73	0.72	0.74	0.55	0.69	0.57
J48 Unpruned	**0.81**	0.42	0.53	0.73	0.66	0.74	0.55	0.69	0.54
J48 Reduced Error Pruning	**0.81**	0.25	0.56	0.72	0.68	0.74	0.55	0.69	0.55
J48 Pruned	0.75	0.42	0.58	0.73	0.65	0.73	0.55	0.69	0.54
C-SVC RBF	**0.81**	0.40	0.61	0.72	0.66	0.72	0.56	0.69	0.54
ν-SVC RBF	0.77	**0.56**	0.72	0.74	0.70	0.76	0.56	0.69	0.58

Legend: Acc: Accuracy, F_1: F1-measure, AUC: area under ROC.

- 影响结果的因素：
  - 是否使用聚类
  - 聚类算法的种类
  - 数据集的大小

对有效性的威胁

1. 外部的威胁

- 训练集的标注具有主观性
- 基于Joiner的召回率100%的假设来评估classifier，这一点没有被完全验证
- 数据集不够大

2. 内部的威胁

- MARPLE不能处理引用的类库

结论和未来工作

1. 结论

- 实验表明，应用机器学习可以显著提高探测四种模式的表现。对于组合模式则提高不显著
- 探测单级模式的效果优于探测多级模式

- 组合模式准确率低，作者认为是数据集太小的原因

2. 将来工作

- 将来希望增加探测的设计模式和软件系统的数量
- 将来希望使用更多的软件包进行机器学习的计算
- 将来希望能比较本工具和其他文献中的工具