高级软件工程

软件维护

沈立炜

shenliwei@fudan.edu.cn

软件的持续演化

- **软件首次交付之后…**
  - 软件中隐藏的错误和缺陷可能陆续被发现
  - 新的用户需求可能不断被提出
  - 新的开发技术会不断涌现
  - 系统的软硬件运行环境也可能不断发生变化
  - 新的市场机会要求基于已有产品开发变体产品
- **增量、迭代的软件开发过程（如敏捷方法）：软件演化进一步与软件开发过程本身融合在了一起**

软件维护

- **通过多种技术和管理手段，高效、高质量地实现各种软件演化目标**
  - 程序理解
  - 任务管理
  - 逆向工程
  - 软件重构
  - 代码辅助完成
  - 演化质量保障
  - …

软件维护类型

- 纠正性维护（corrective maintenance）：针对软件交付后所发现的问题（如软件错误），为纠正和解决问题使其符合软件需求而对软件进行的修改
- 预防性维护（preventive maintenance）：为改进软件的可维护性和可靠性、避免今后可能出现的各种问题而对软件进行的修改
- 适应性维护（adaptive maintenance）：为使得软件能够适应发生变化的运行环境（例如操作系统、网络环境等）而对软件进行的修改
- 完善性维护（perfective maintenance）：为完善、改进和增强软件的功能以及性能、可靠性等质量属性而对软件进行的修改

软件维护成本

- 许多研究都表明软件维护已经占到整个软件开发成本的60%-70%甚至更多
- 其中纠正性维护（含预防性维护）所占的成本只有20%左右，其他大多数都是用于非纠正性维护（即适应性维护、完善性维护及其他）

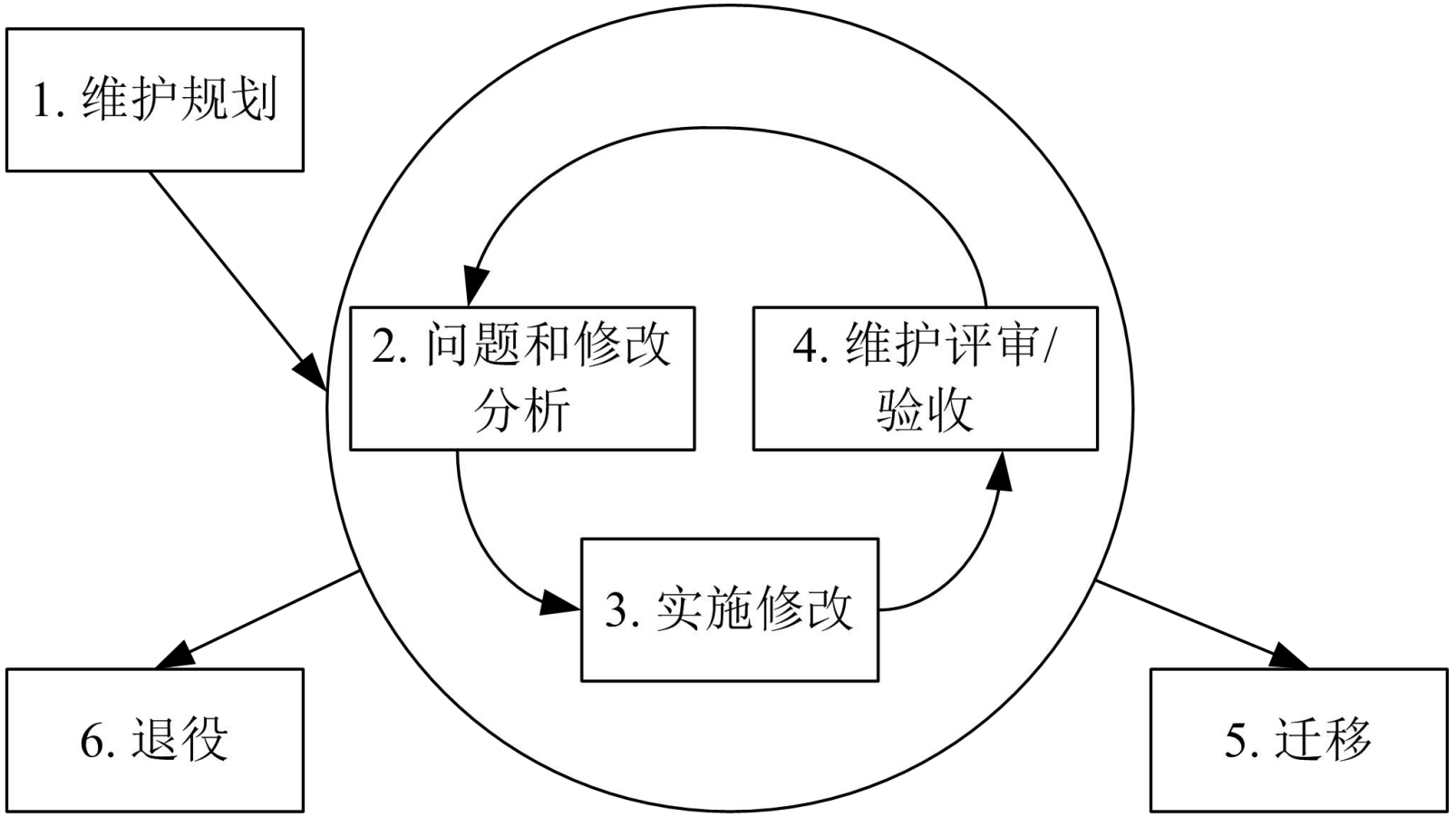
可维护性(MAINTAINABILITY)

- 可维护性：一个软件系统能够在多大程度上被容易地修改以满足各种软件维护和演化目的
  - 可理解性
  - 可测试性
  - 可修改性
  - 可扩展性
  - 可复用性
  - …

影响可维护性的因素

- **软件的可维护性与软件开发过程以及各个开发活动都有关系**
  - 良好的需求与源代码的追踪关系管理
  - 软件设计的模块化程度
  - 编码过程中的代码注释质量、代码复杂度
  - 测试用例的数量、质量以及测试自动化程度
  - 版本管理和自动化构建能力
- **应从一开始就重视软件的可维护性问题，并通过各种手段获得或保持一个软件系统所需的高可维护性**

软件维护过程

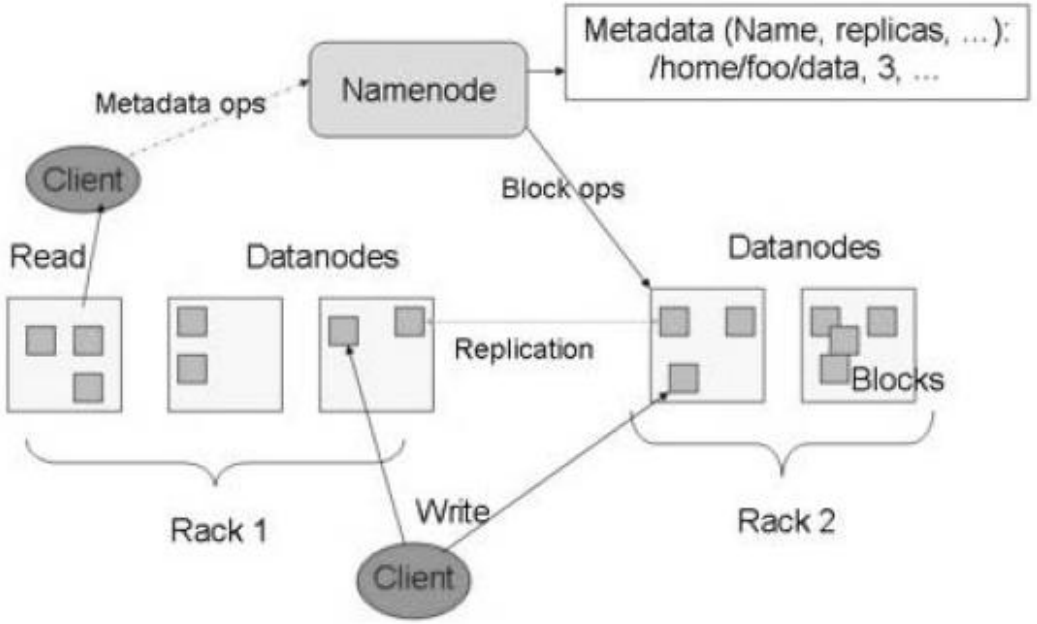


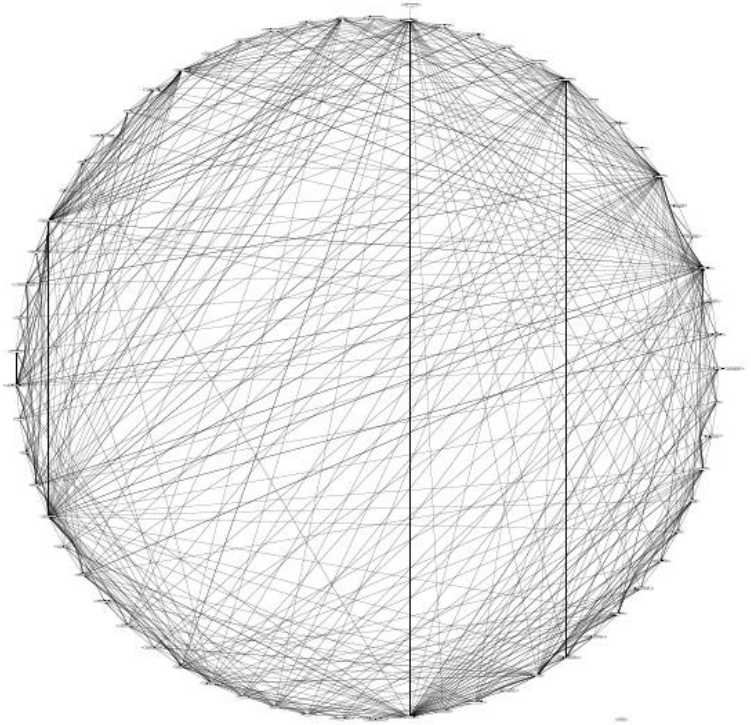
ISO/IEC/IEEE 14764-2996标准（软件工程—软件生存周期过程—维护）

软件维护的冰山

- **软件维护的“冰山”**
  - 时间和成本上的限制
  - 预见性不足导致的不适应
  - 文档缺失或未能同步更新
  - 人员变动
  - 缺少责任意识(过渡心理）
- **技术债(technical debt)**

设计退化

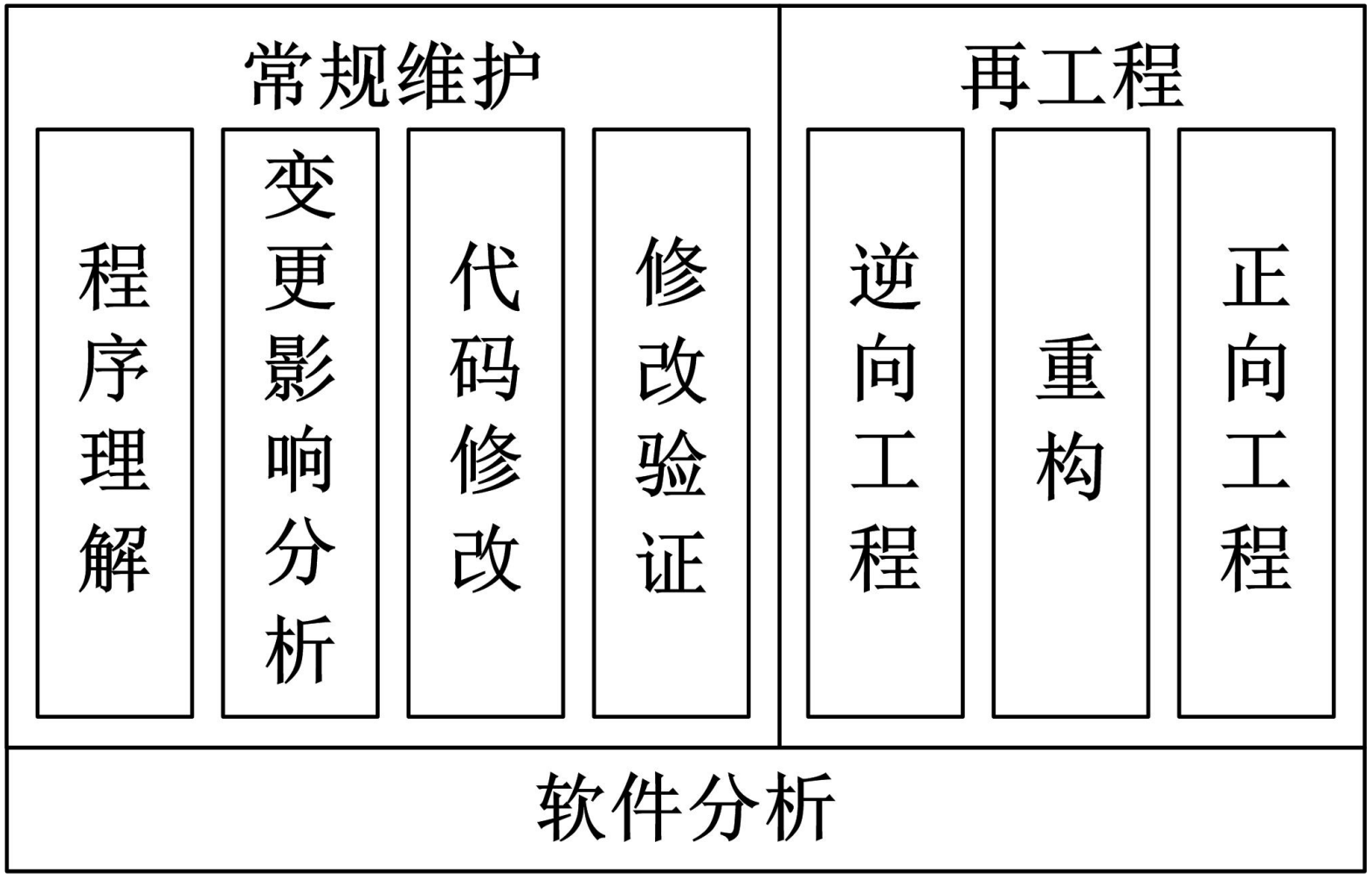




Hadoop文档中的设计结构　　**Hadoop的实际实现结构**

Nenad Medvidovic. Adapting Our View of Software Adaptation: An Architectural Perspective. SEAMS'14, June 2-3, 2014, Hyderabad, India.

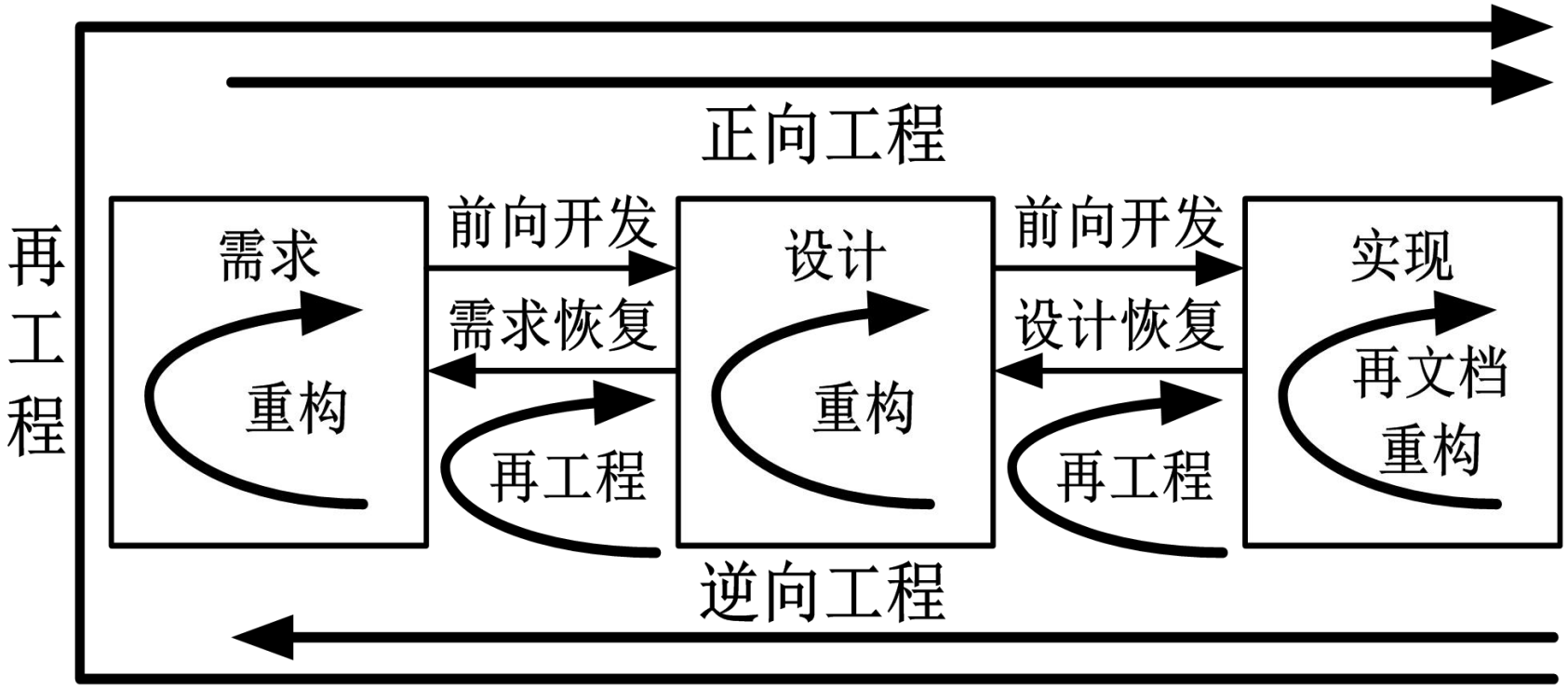
软件维护技术内容



软件再工程

- **软件演化问题的累积**
  - 导致常规维护难以进行
  - 很多软件资产（如构件、代码等）仍然具有利用价值
- **软件再工程是一个工程过程，它将逆向工程、结构重组和正向工程组合起来，将现存系统重新构造为新的形式**
  - 改善系统的结构
  - 消除潜在的缺陷
  - 延长系统的生命周期
  - 同时又能充分利用遗产系统，节约时间和成本

软件再工程过程



软件分析

- 软件分析是对软件进行人工或者自动分析，以验证、确认或发现软件性质（或者规约、约束）的过程或活动
- 软件分析的对象：源代码、目标代码、文档、模型、开发历史等

软件分析类型

- **静态分析**
  - 分析对象主要是源代码、目标代码和开发文档等
  - 分析过程中不需要系统进行运行
- **动态分析**
  - 目标是获取系统的动态运行信息
  - 一般通过程序插装向代码中植入与动态信息收集相关的额外代码，然后通过测试用例驱动的方式运行系统获得动态信息
- **开发历史分析**
  - 对软件开发信息库（如版本库、缺陷库、问题追踪系统、邮件列表等）中反映的开发历史进行分析和挖掘

静态分析-1

- 基本分析：包含一些常见的基础性分析技术，例如语法分析、类型分析、控制流分析、数据流分析等，大多数编译器本身都包含这些分析过程
- 基于形式化方法的分析：采用一些数学上比较成熟的形式化方法，通过分析获得关于代码的一些更精确或者更广泛的性质，包括定理证明、模型检测、抽象解释、约束求解等

静态分析-2

- 指向分析：这类分析多数与指针密切相关，包括别名分析、指针分析、形态分析、逃逸分析等，其目的确定指针及内存引用等的指向从而提高基本分析（如数据流分析）的精度
- 其它辅助分析：包含了其他一些辅助性的分析技术，可以为前面几类分析提供辅助性的支持，包括符号执行、切片分析、结构分析、克隆分析等

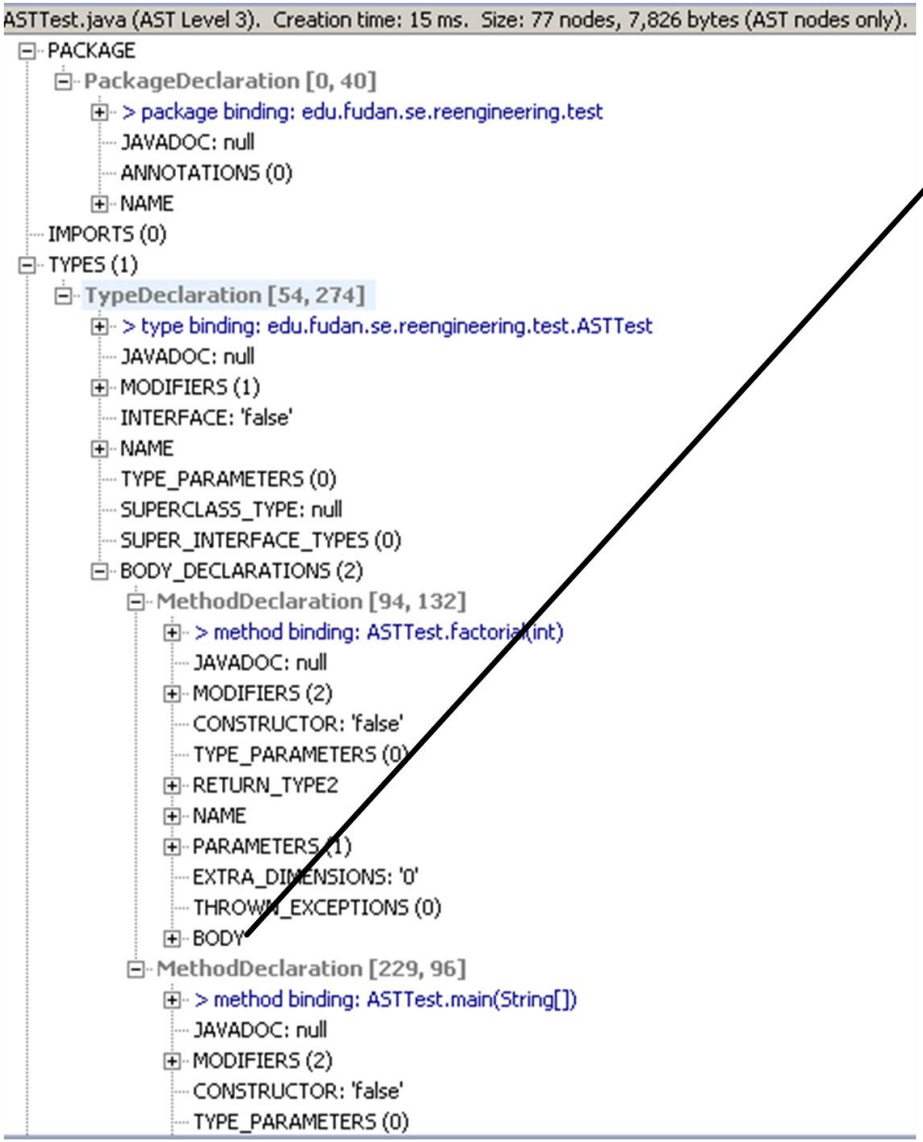
语法分析

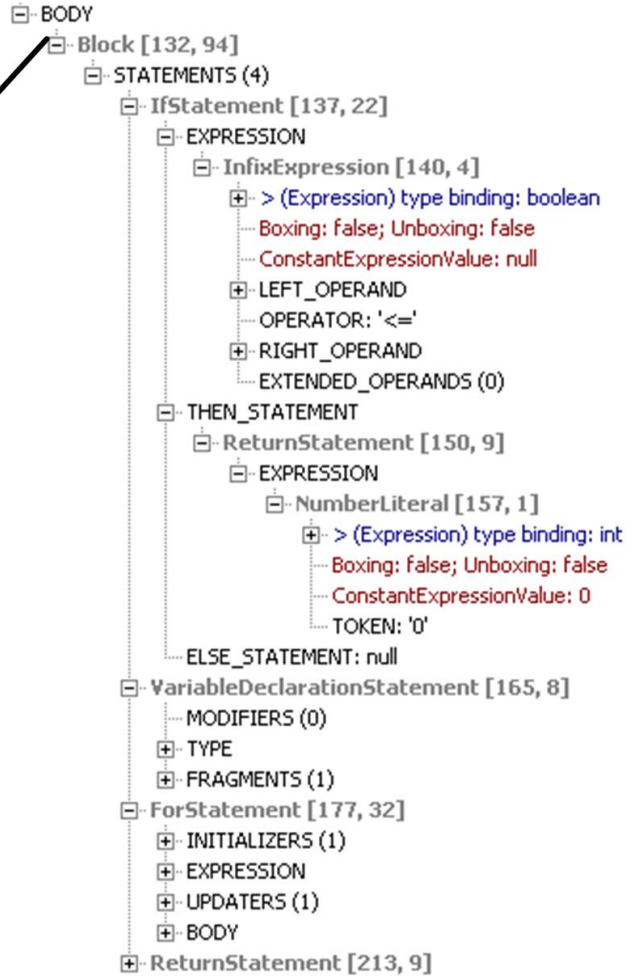
- 表示：抽象语法树（Abstract Syntax Tree，简称AST）
- 抽象语法树是许多程序分析技术的基础

语法分析-示例代码

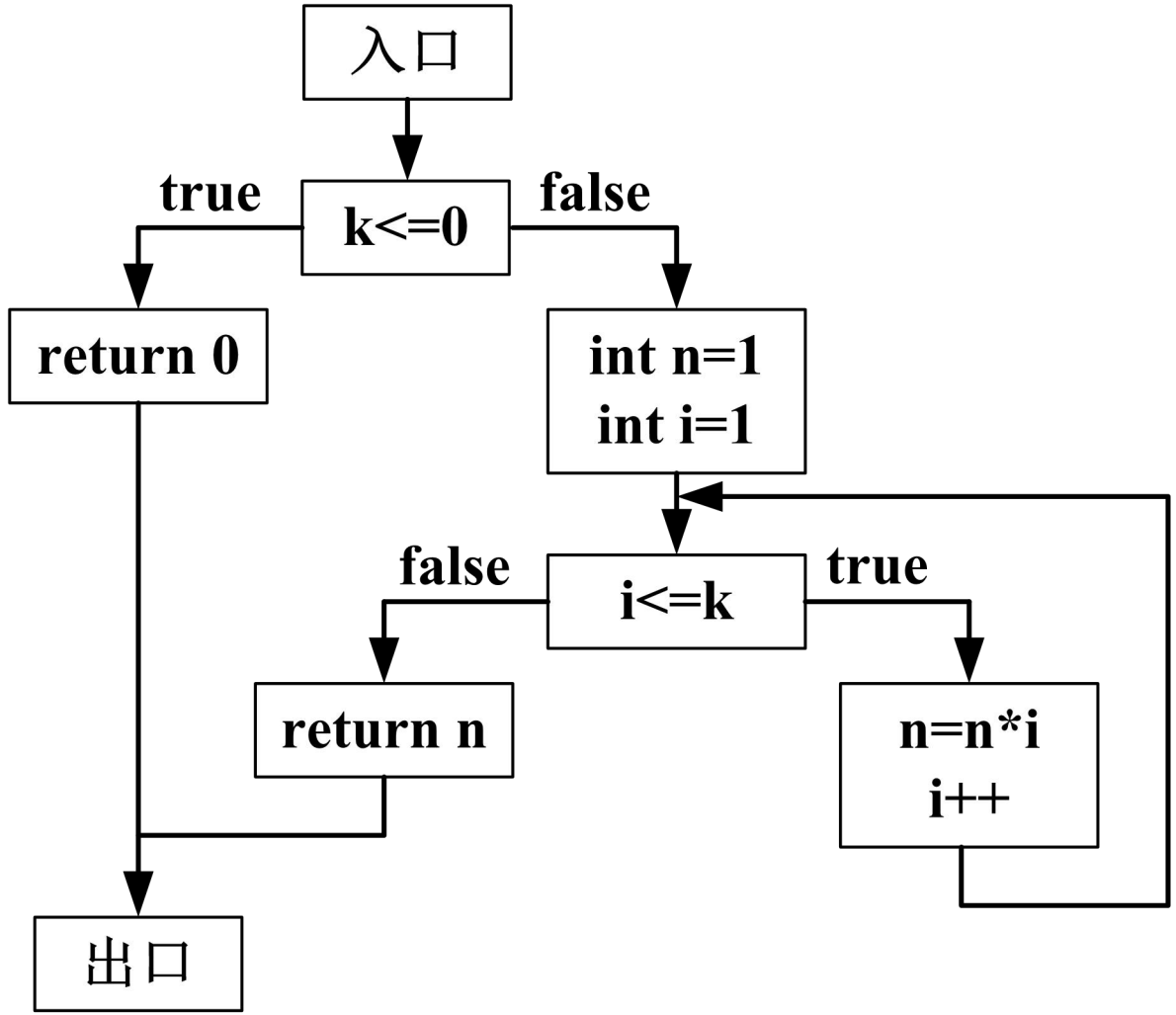
```
package edu.fudan.se.reengineering.test;
//Java 实例程序
public class ASTTest
{
 //求阶乘 k!的函数
 public static int factorial (int k)
 {
  if(k<=0)
   return 0;
  int n=1;
  for (int i=1;i<=k;i++)
   n=n*i;
  return n;
 }
 public static void main (String args [ ])
 {
  int n=factorial (10);
  System.out.println (n);
 }
}
```

语法分析-示例语法树


ASTTest.java (AST Level 3). Creation time: 15 ms. Size: 77 nodes, 7,826 bytes (AST nodes only).
PACKAGE
PackageDeclaration [0, 40]
> package binding: edu.fudan.se.reengineering.test
JAVADOC: null
ANNOTATIONS (0)
NAME
IMPORTS (0)
TYPES (1)
TypeDeclaration [54, 274]
> type binding: edu.fudan.se.reengineering.test.ASTTest
JAVADOC: null
MODIFIERS (1)
INTERFACE: 'false'
NAME
TYPE_PARAMETERS (0)
SUPERCLASS_TYPE: null
SUPER_INTERFACE_TYPES (0)
BODY_DECLARATIONS (2)
MethodDeclaration [94, 132]
> method binding: ASTTest.factorial(int)
JAVADOC: null
MODIFIERS (2)
CONSTRUCTOR: 'false'
TYPE_PARAMETERS (0)
RETURN_TYPE2
NAME
PARAMETERS (1)
EXTRA_DIMENSIONS: '0'
THROWN_EXCEPTIONS (0)
BODY
MethodDeclaration [229, 96]
> method binding: ASTTest.main(String[])
JAVADOC: null
MODIFIERS (2)
CONSTRUCTOR: 'false'
TYPE_PARAMETERS (0)


BODY
Block [132, 94]
STATEMENTS (4)
IfStatement [137, 22]
EXPRESSION
InfixExpression [140, 4]
> (Expression) type binding: boolean
Boxing: false; Unboxing: false
ConstantExpressionValue: null
LEFT_OPERAND
OPERATOR: '<='
RIGHT_OPERAND
EXTENDED_OPERANDS (0)
THEN_STATEMENT
ReturnStatement [150, 9]
EXPRESSION
NumberLiteral [157, 1]
> (Expression) type binding: int
Boxing: false; Unboxing: false
ConstantExpressionValue: 0
TOKEN: '0'
ELSE_STATEMENT: null
VariableDeclarationStatement [165, 8]
MODIFIERS (0)
TYPE
FRAGMENTS (1)
ForStatement [177, 32]
INITIALIZERS (1)
EXPRESSION
UPDATERS (1)
BODY
ReturnStatement [213, 9]

控制流分析



factorial方法的控制流图

数据流分析

- 关注于语句或代码块之间的数据（体现为变量）依赖关系：变量定值、注销定值、以及定值的传入传出
- 求解每个代码块中所用变量的数据依赖关系（即所引用的变量的值依赖于前面哪些程序语句），用于：
  - 通过变量的定义和加工处理过程理解程序的计算过程
  - 对程序中的数据流异常进行检查：变量无定值使用、变量重复定值、变量定值无使用

程序依赖图

- **PDG：Program Dependence Graph**
- **基于控制流和数据流分析**
  - 控制依赖
  - 数据依赖
- **为进一步的程序分析（如功能定位、克隆分析等）打下基础**

程序切片(PROGRAM SLICING)

- 问题：计算机程序在程序理解和调试等方面的复杂性很大程度上是由于多种处理和计算逻辑混杂在一起
- 手段：将程序中与指定的变量或数据结构相关的部分抽取出来，从而将目标关注范围局限到较小的范围内，帮助程序理解和调试等

程序切片的定义

对于程序P中某个感兴趣的程序点（一般是某一程序行）n上的计算中所用到的变量V，P中那些可能影响该处计算中V的取值的那些程序语句构成了一个**程序片（Program Slice）**，其中（n, V）称为**切片标准**。

程序切片示例

```
1. read x;
2. y=x*2;
3. if(y>0){
4.   x=x+5;
5.   y=y*2;
6. }
7. else
8.   x=x+10;
9. write y;
```

源程序

```
1. read x;
2. y=x*2;
3. if(y>0){

5.   y=y*2;
6. }
```

针对(9,y)的切片

程序切片的分类

- 根据切片是否可执行：可执行切片和不可执行切片
- 根据切片方向：前向切片和后向切片
- 根据切片范围：过程内切片和过程间切片
- 根据切片分析时间：静态切片和动态切片

克隆分析

- **代码克隆：源代码文件中多个相同或相似的代码片断**
  - 有意识的克隆：复制/粘贴式的代码复用
  - 偶然克隆：设计思想以及编码习惯上的相似性
- **代码克隆大量存在于单个软件系统以及若干相似的软件系统中**
  - 例如，JDK 1.3.0中有40%多的代码文件和20%多的代码行中存在不同程度的代码克隆

产生克隆的原因

- **有意识的克隆：往往由所采用的开发和维护策略或者语言及开发技术上的局限性造成的**
  - 例如通过代码复制粘贴实现复用、生成式编程、相似系统的合并、为避免风险或更好的性能而进行的代码复制等
- **偶然克隆：与模式化的编程风格相关**
  - 例如调用相同的程序库和API时的相似编程模式、不同程序员实现相似功能时的相似代码等

代码克隆的有害性

- **代码克隆在大多数情况下是有害的**
  - 增加了代码长度，导致软件理解和维护的负担
  - 克隆代码分散在不同地方，带来一致性修改的问题
- **代码克隆的“有利”方面**
  - 简单有效的代码复用手段
  - 保持模块的独立性
  - 学习和探索技术方案的途径
- **克隆管理：无论是否有害，都要对代码克隆进行有效的管理**

代码克隆侦测方法

- 基于文本（text）的方法：将代码看作文本，以代码文本上的相似性作为判断代码相似性的依据
- 基于标号（token）的方法：按照编程语言的词法规则将代码转换为标号序列，通过后缀树算法识别相似片段
- 基于抽象语法树（AST）的方法：在AST基础上通过子树相似度计算来发现克隆片断
- 基于度量（metrics）的方法：将代码划分为固定规模的单元，然后根据代码单元之间度量值（例如扇入、扇出、变量信息等）的相似性发现克隆片段
- 基于程序依赖图（PDG）的方法：在程序依赖图基础上寻找同构的子图

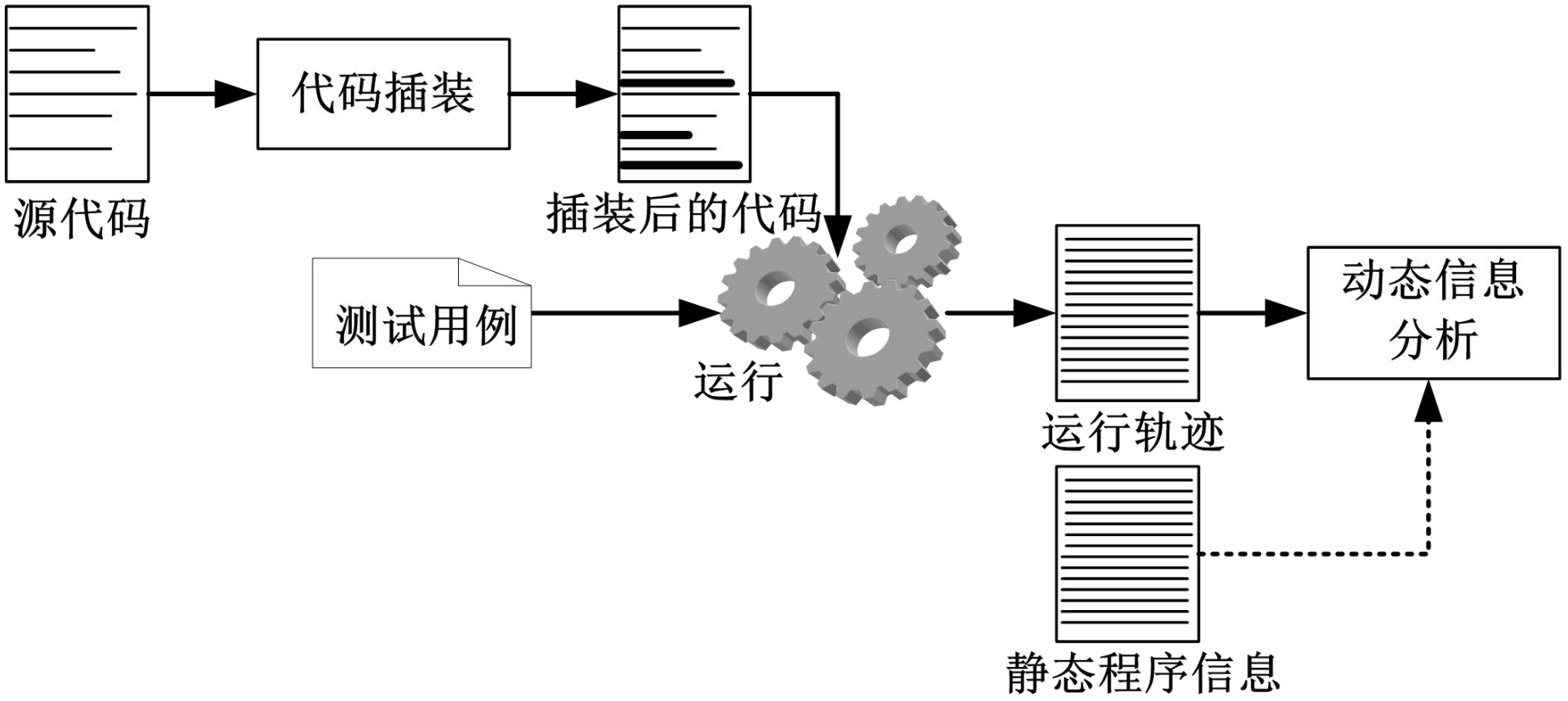
程序文本分析

- 程序中的文本信息：类、函数和变量命名、代码注释等，可以提供额外的语义信息（例如与该段代码相关的需求或设计概念等）
- 将信息检索、文本处理等方法应用到程序文本和相关开发文档信息的分析中去，分析程序中所蕴含的高层概念
  - 向量空间模型（Vector Space Model，简称VSM）
  - 隐性语义索引（Latent Semantic Indexing，简称LSI）
- 用于抽取代码主题及关注点、实现高层概念（需求、设计）与代码的追踪关系计算、进行代码语义聚类等

程序动态分析

- 通过测试用例驱动的方式对系统进行运行，并通过获取的系统动态运行信息分析系统的设计和需求信息
- 虽然具有代价高、不完整的弱点，但它能够弥补静态分析的许多不足
  - 可以很容易地获取各种系统交互信息
  - 面向对象技术中继承、动态绑定等机制，以及分布式计算所带来的不确定性

动态分析基本过程



动态分析的典型应用

- **面向对象程序的UML顺序图逆向恢复**
  - 顺序图由于与程序的动态交互信息相关，因此很难通过静态分析获得
  - 动态分析可以很容易获取特定场景下各个对象之间的消息发送顺序等交互信息，而且可以区分不同的类实例对象
  - 每次运行获得的交互信息可以构成一个顺序图的交互实例
  - 在此基础上结合静态语法信息（例如位于同一if语句不同分支的交互消息可以归并为顺序图中的选择片断）可以获得完整的顺序图

动态分析的困难

- **主要难点是代码插装和测试用例设计**
- **传统的代码插装技术需要在语法分析基础上针对源代码或目标代码进行，**
  - 缺点：比较复杂且监控代码与源代码紧密耦合
- **基于反射技术、开放编译以及AOP等技术的代码插装**
- **主要缺点**
  - 代码插装和测试用例设计带来的额外成本
  - 动态分析的不完整性

开发历史分析

- **分析对象是与开发过程相关的信息库：版本控制系统（如CVS）、bug追踪系统（如Bugzilla）、变更管理和过程管理系统等**
- **开发库记录了软件开发和演化过程的完整轨迹，是对静态和动态分析的有效补充**
  - 不同版本之间的差异
  - 同一时间段内一起被修改的文件之间的变更耦合关系
  - bug报告与代码库之间的关系
  - 问题解决报告和CVS消息中所记录的修改说明等

开发历史分析的应用

- **通过数据挖掘、机器学习等方法分析开发历史数据**
- **典型应用**
  - bug报告分派（由谁负责、可能涉及哪些文件）
  - 发现相似问题解决方案
  - 从演化的角度分析耦合关系
  - 缺陷预测和缺陷侦测

程序理解和变更影响

- 程序理解：执行软件维护任务的开发人员在进行代码修改之前首先理解相关程序的功能、结构和行为等方面，从而为确定修改方案打下基础
- 变更影响分析：分析并确定代码修改的影响范围

程序理解和变更影响

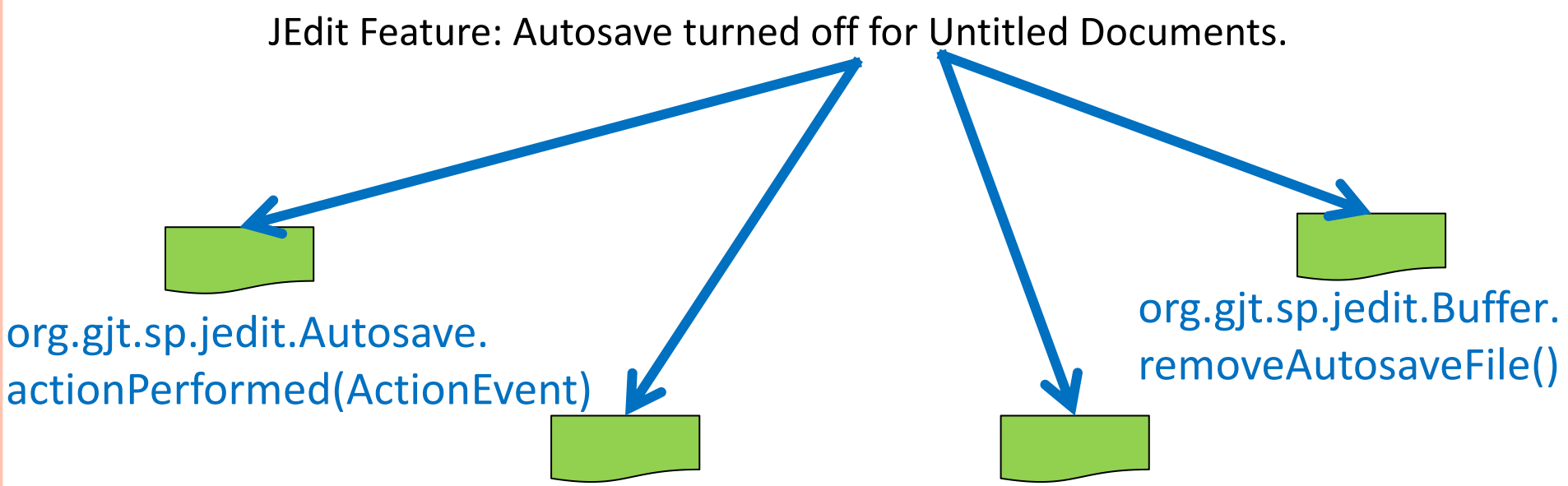
- 程序理解：执行软件维护任务的开发人员在进行代码修改之前首先理解相关程序的功能、结构和行为等方面，从而为确定修改方案打下基础
- 变更影响分析：分析并确定代码修改的影响范围

特征定位

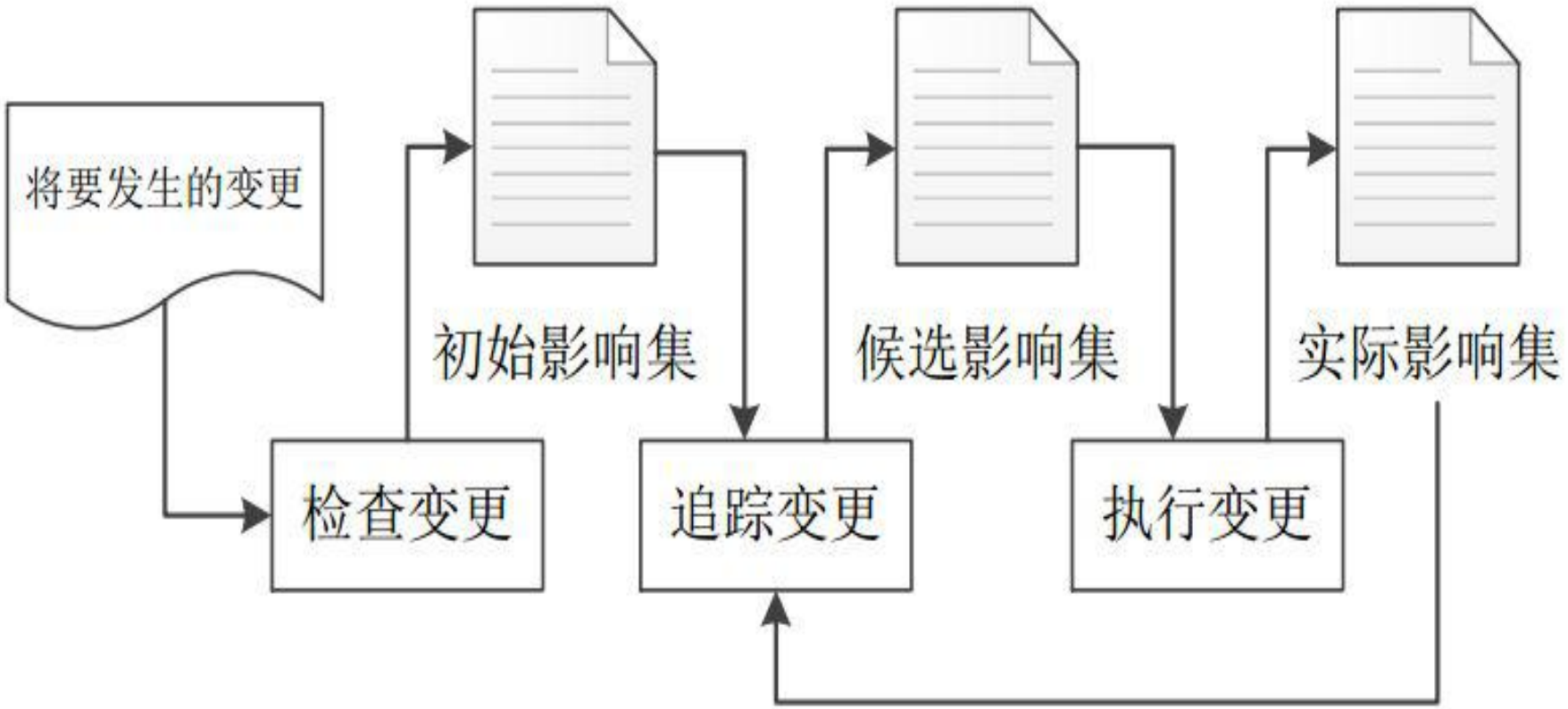
- 软件维护任务经常是根据用户可见的外部功能特征来刻画和布置的，例如
  - 修复一个已有特征实现代码中的缺陷
  - 在一个已有特征基础上增加一个新的子特性等
- 开发人员在确定修改方案之前必须首先确定与给定特征相关的代码单元（如类、方法等）的位置并理解其实现方式，这一过程被称为**特征定位**

特征定位示例

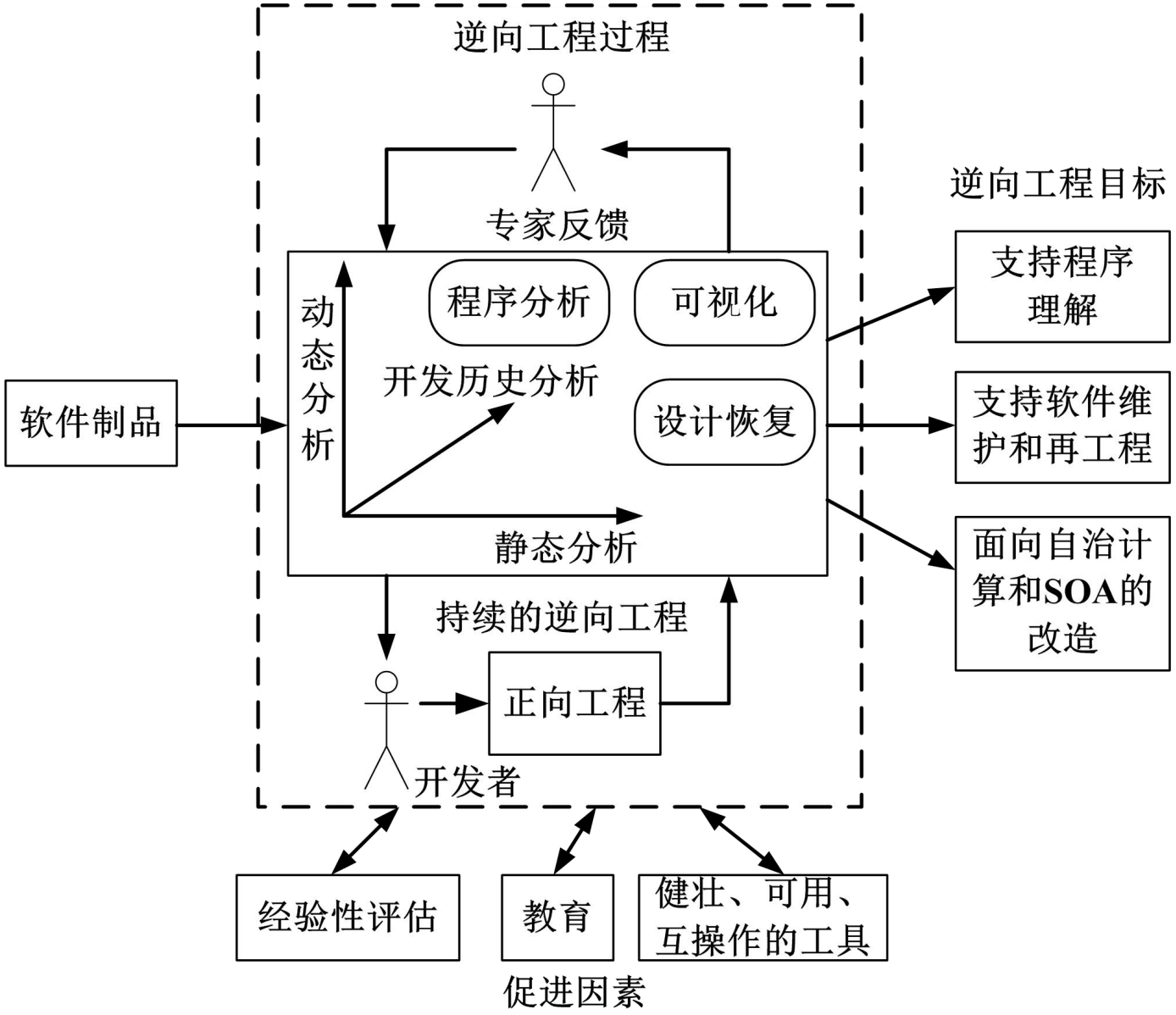
在代码中定位与给定特征相关的实现单元：确定一个用户可见的需求特征在代码中的哪些地方以及如何实现



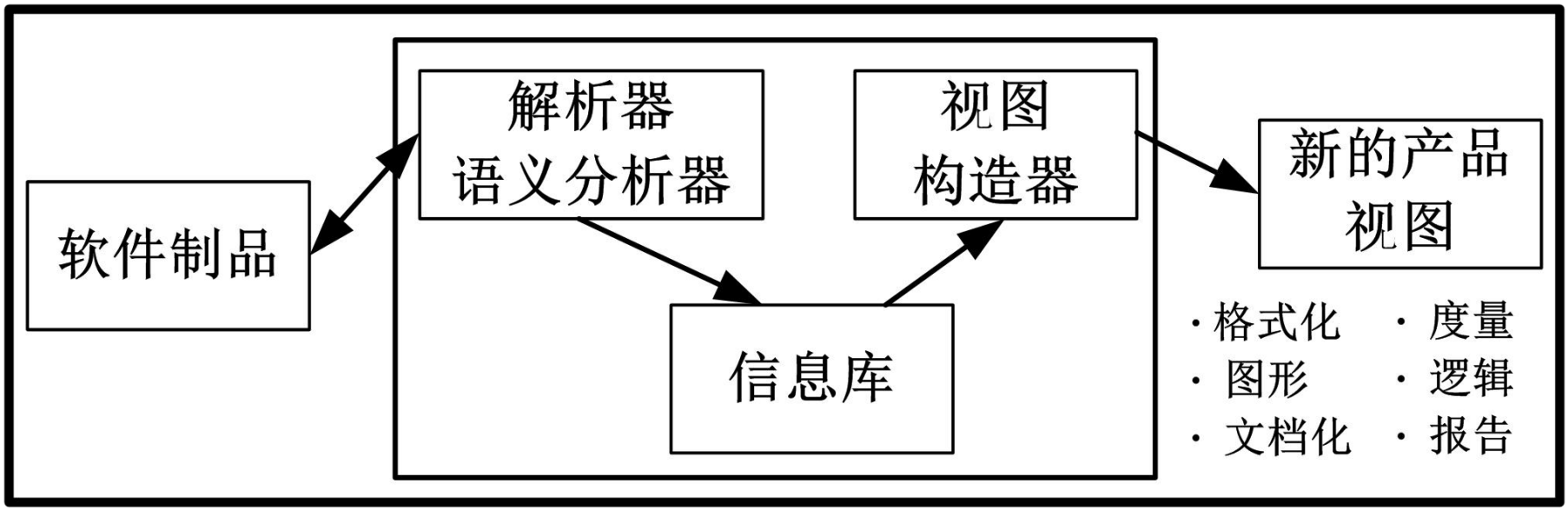
变更影响分析过程



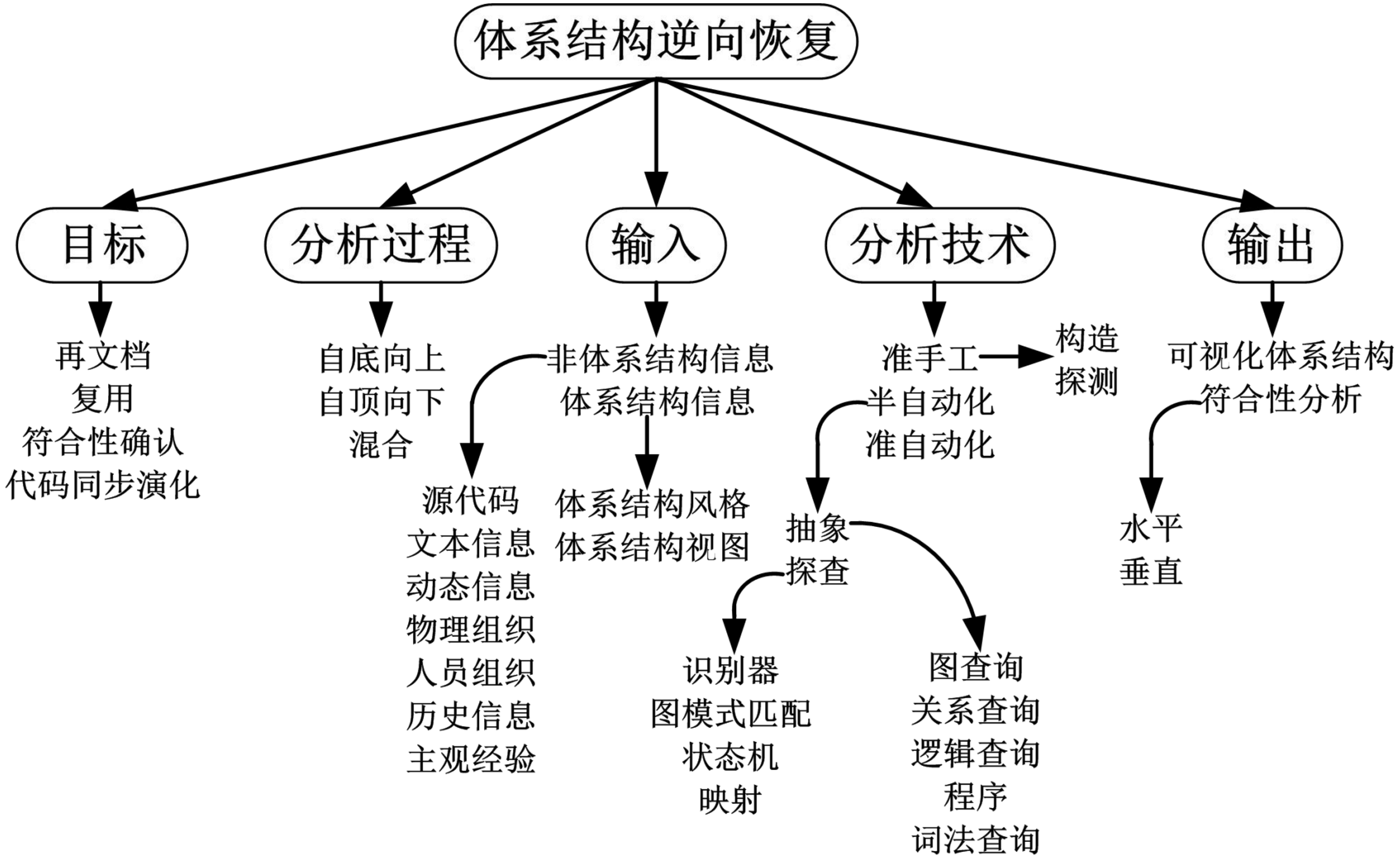
软件逆向工程的角色


逆向工程过程
专家反馈
逆向工程目标
程序分析
可视化
动态分析
开发历史分析
设计恢复
软件制品
支持程序理解
支持软件维护和再工程
面向自治计算和SOA的改造
静态分析
持续的逆向工程
正向工程
开发者
经验性评估
教育
健壮、可用、互操作的工具
促进因素

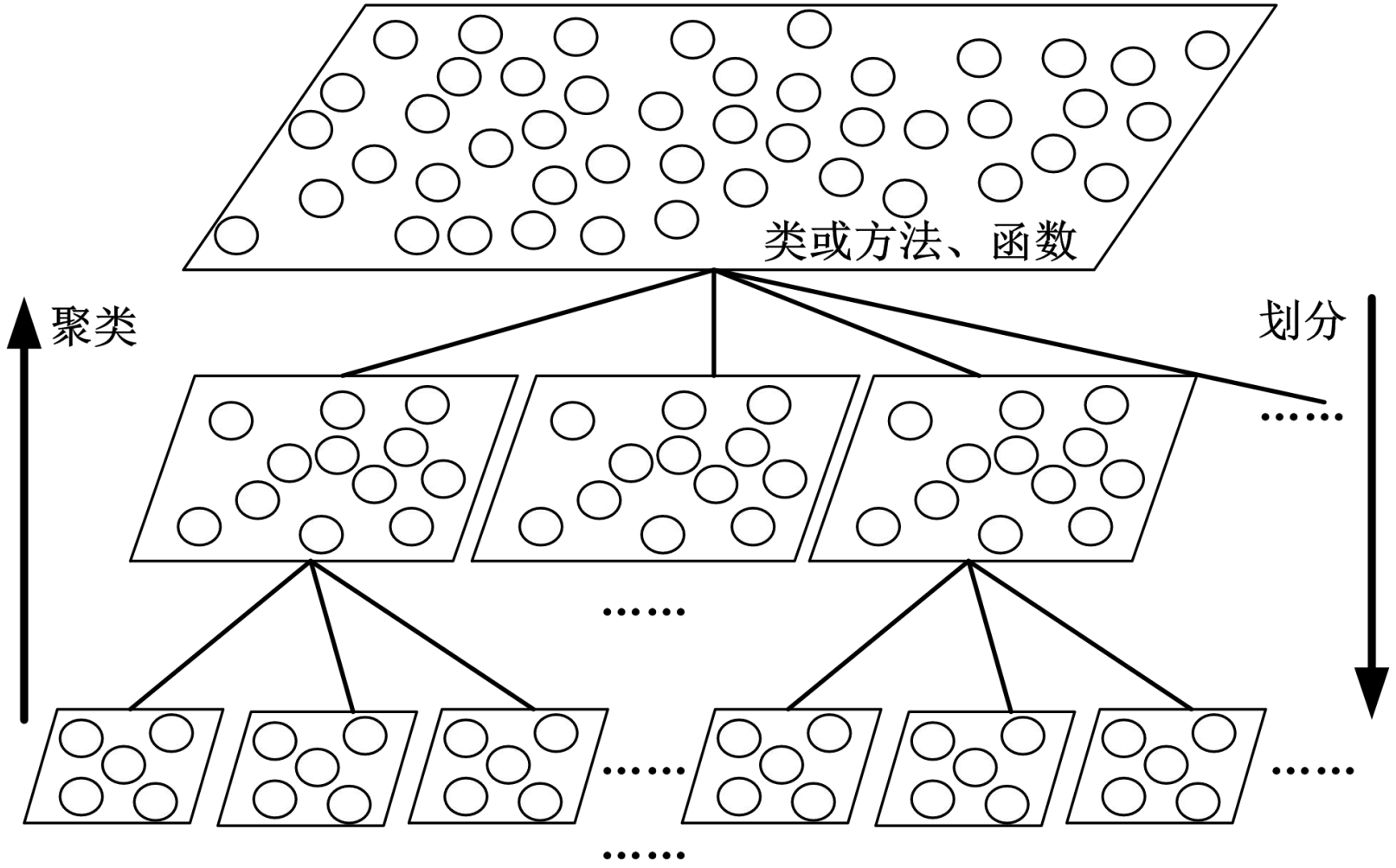
逆向工程工具的一般架构



体系结构逆向恢复方法

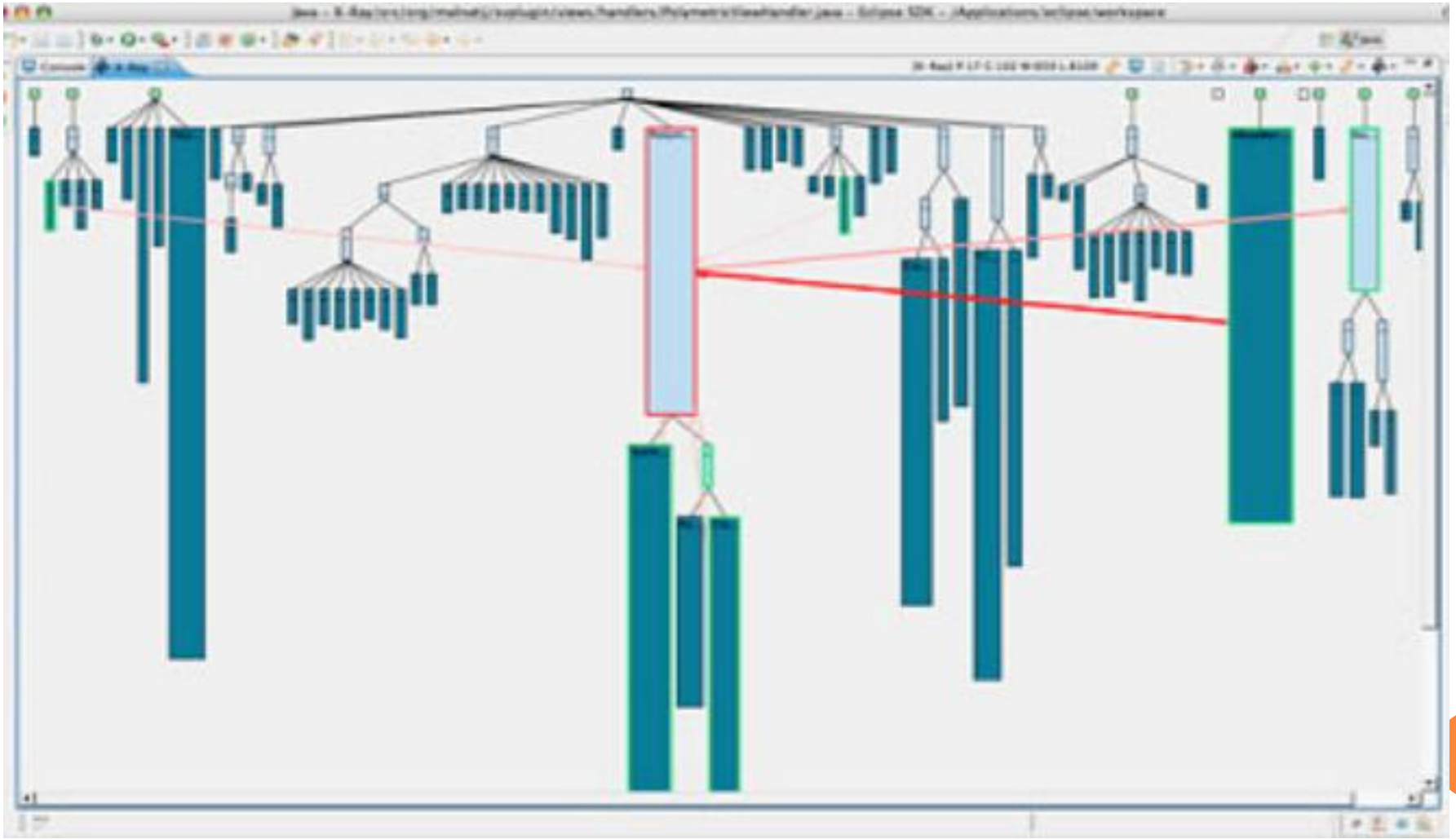

体系结构逆向恢复
目标
分析过程
输入
分析技术
输出
再文档
复用
符合性确认
代码同步演化
自底向上
自顶向下
混合
非体系结构信息
体系结构信息
源代码
文本信息
动态信息
物理组织
人员组织
历史信息
主观经验
体系结构风格
体系结构视图
准手工
构造
探测
半自动化
准自动化
抽象
探查
识别器
图模式匹配
状态机
映射
图查询
关系查询
逻辑查询
程序
词法查询
可视化体系结构
符合性分析
水平
垂直

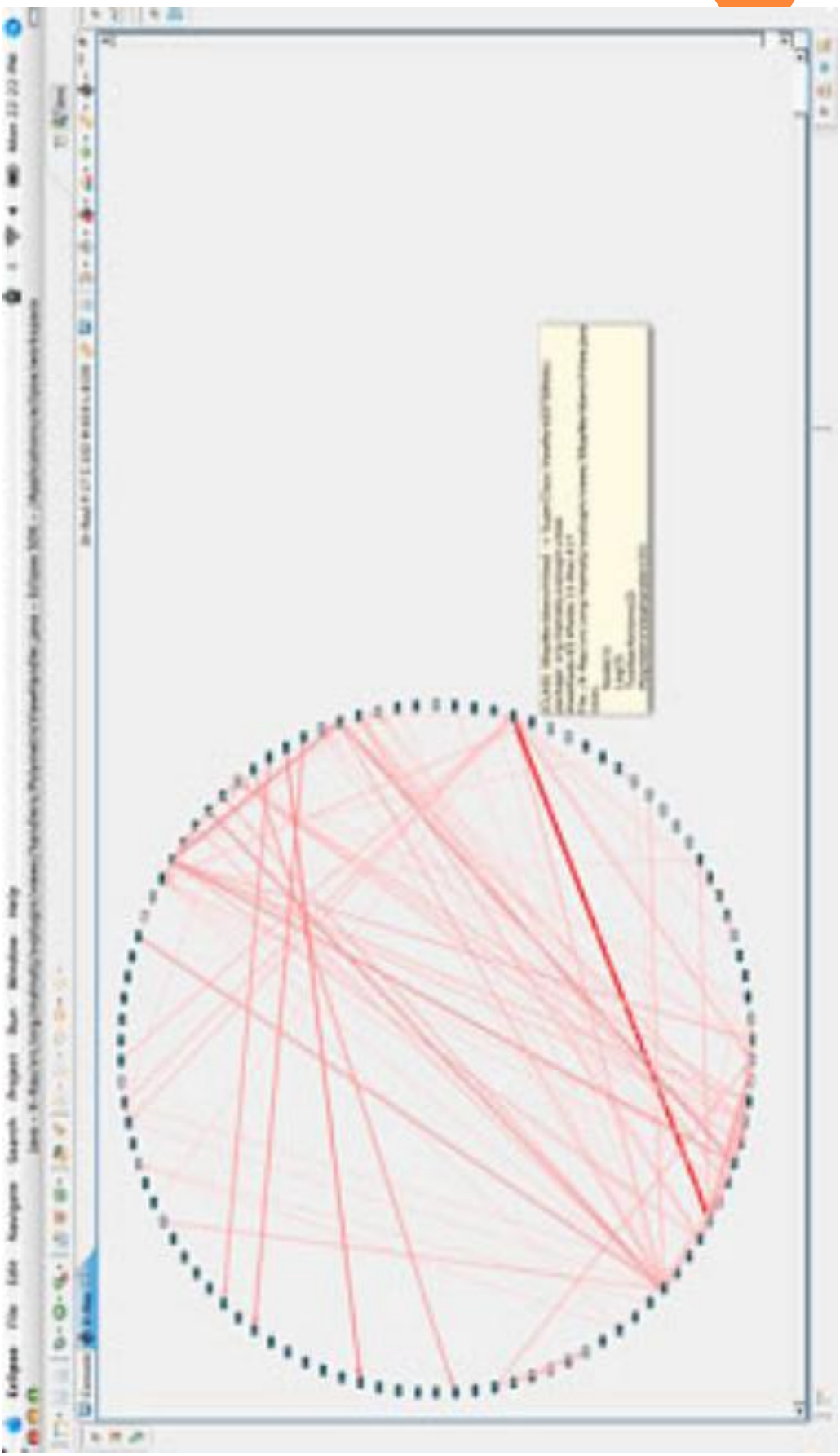
基于聚类/划分的体系结构恢复

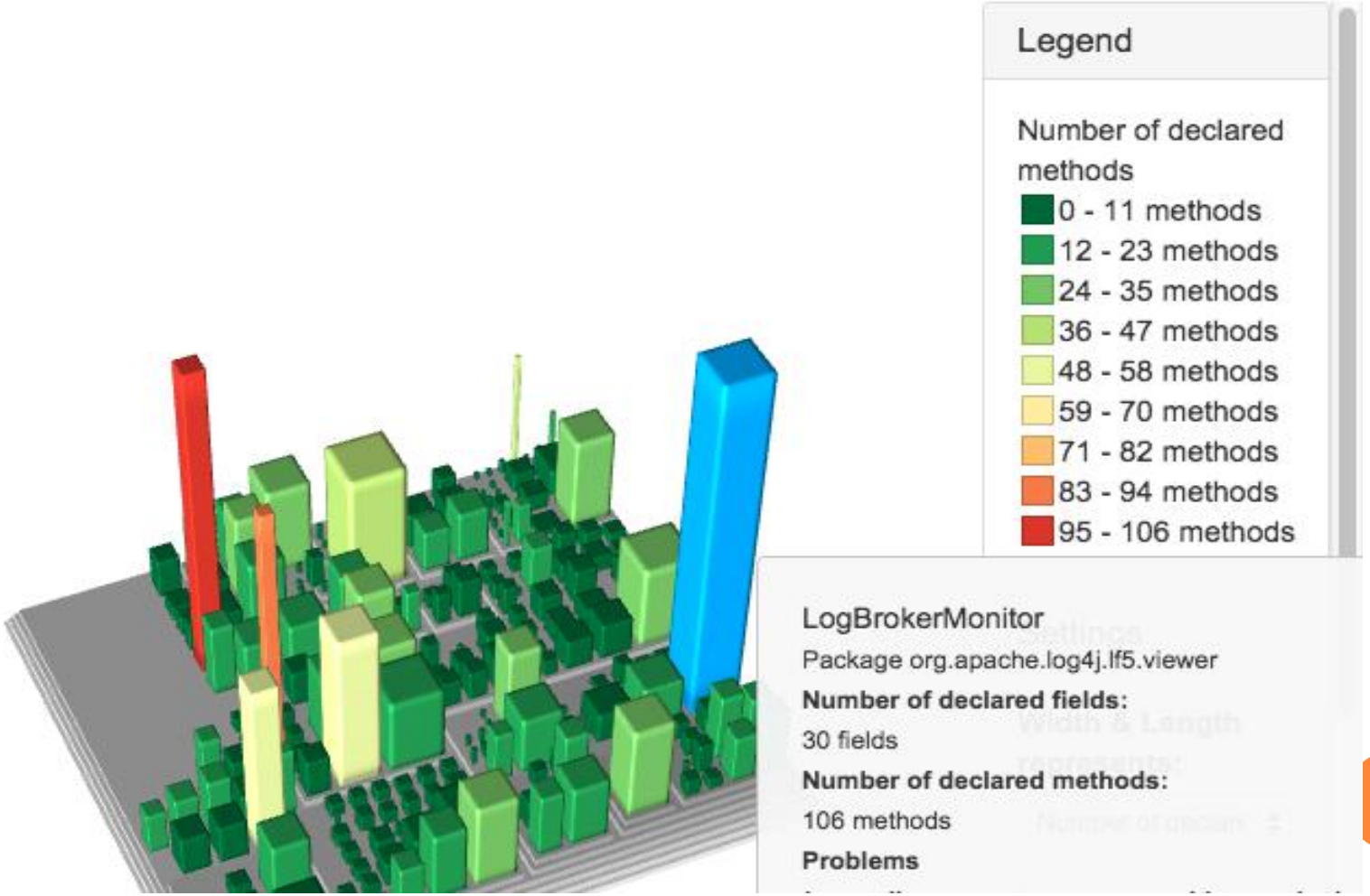


- 基于反射的体系结构逆向恢复
- 用况驱动的体系结构恢复方法
- 基于体系结构风格的运行时体系结构恢复方法

软件可视化






Legend
Number of declared methods
0 - 11 methods
12 - 23 methods
24 - 35 methods
36 - 47 methods
48 - 58 methods
59 - 70 methods
71 - 82 methods
83 - 94 methods
95 - 106 methods
LogBrokerMonitor
Package org.apache.log4j.lf5.viewer
Number of declared fields:
30 fields
Number of declared methods:
106 methods
Problems

软件重构

- 软件重构是指在不改变程序外部行为的情况下对其内部设计和实现进行改进和优化的一种软件维护实践

代码的坏味道

- 软件的实现代码中经常会集聚越来越多影响软件可维护性的问题，这些问题被人们形象地形容为代码中的坏味道
- 软件重构可以被理解为识别代码中的坏味道并通过各种手段缓解或消除其影响的过程
- Martin Fowler. 《Refactoring: Improving the Design of Existing Code》

- **膨胀**
  - 长方法
  - 过大的类
  - 基本类型偏执
  - 长参数列表
  - 数据泥团

面向对象的滥用

- Switch语句
- 临时属性
- 被拒绝的遗赠（Refused Bequest）
- 异曲同工的类

- **变更阻挠者**
  - 发散式变化
  - 霰弹式变化
  - 平行继承体系

- **非必需的内容**
  - 重复代码
  - 冗赘类
  - 数据类
  - 死代码
  - 不必要的通用型

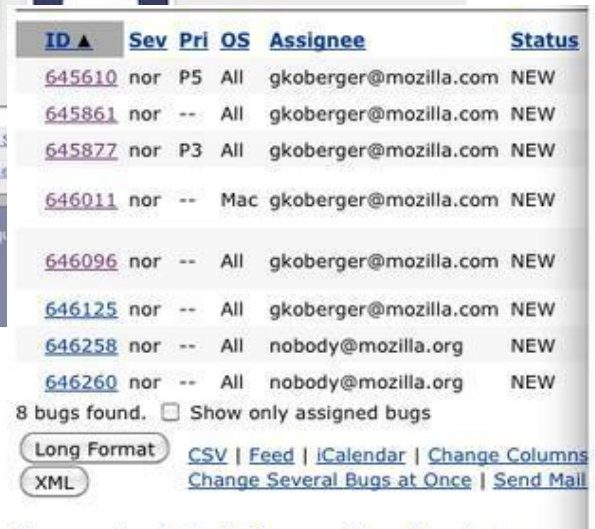
- **耦合者**
  - 特征依恋
  - 狎昵关系（Inappropriate Intimacy）
  - 消息链
  - 中间人

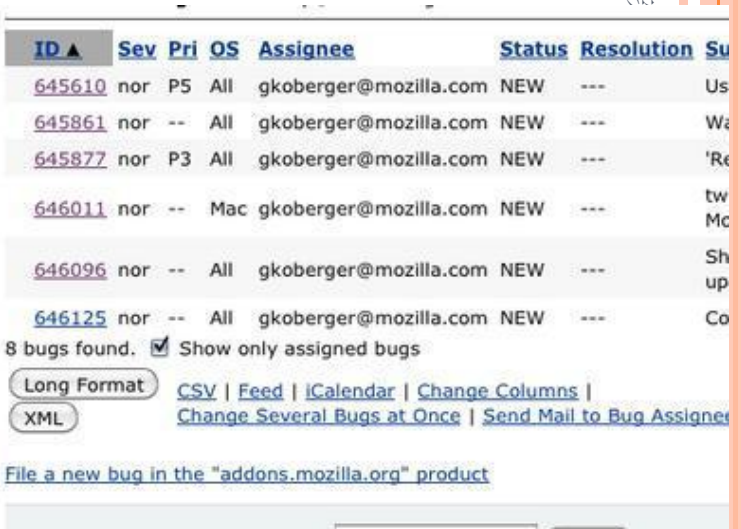
基本的软件重构类型

- 重新组织方法
- 移动成员方法
- 重新组织数据
- 简化条件表达式
- 简化方法调用
- 处理泛化关系
- 大规模重构

软件维护工具

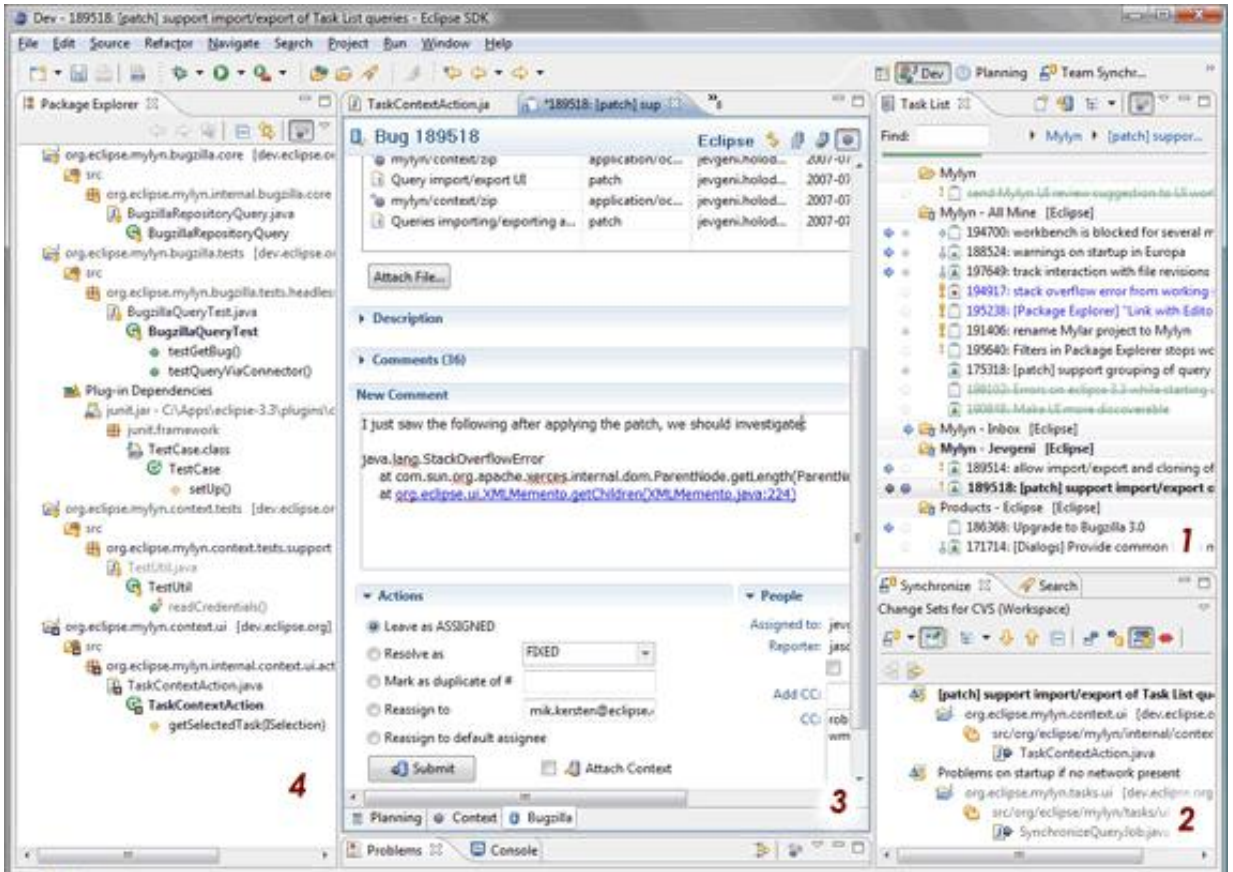
- 缺陷跟踪管理工具
  - BugZilla
  - JIRA





任务管理工具

- MyLyn



- 特征定位工具
- 克隆分析工具
- 逆向分析工具
- 代码分析和度量工具

高级软件工程

软件维护

The End