第十次作业——软件兼容性

课程名称:	软件质量分析	指导教师:	陈仪香
姓　　名:	王海生	学　　号:	10235101559
年　　级:	2023 级	主　　题:	软件兼容性

目录

1 第一题

1.1 题目

开源软件兼容性问题严重程度分成哪 4 级？分别表示什么风险？

1.2 解答

开源软件兼容性问题严重程度分为 `High`、`Medium`、`Low`、`No` 四级，各级别风险如下：

1. **High（高风险）**：表示严重的不兼容变动。例如，类被移除，导致找不到该类和其中的字段或方法；方法可见性降低，找不到该方法；C++ 中类新增纯虚方法，导致该类必须被继承、该纯虚方法必须被重写等。这些问题会对软件的正常运行产生重大影响，可能导致软件编译失败、运行时崩溃或出现非预期的行为。
2. **Medium（中等风险）**：指的是对现有 API 的改动，可能会导致一些功能上的变化，但不会像高风险那样直接导致程序无法运行。比如，方法被 final 修饰符修饰，无法被继承重写，但可以正常被调用；类中的某个字段指针级别发生改变，可能导致类型不兼容或语义发生改变等。这些问题虽不会使软件完全无法运行，但可能影响部分功能的正常使用。
3. **Low（低风险）**：通常涉及那些可能引起语义改变的情况，但不会影响编译或链接过程。例如，类新增了一个父类，可能出现字段或方法的重复；某个常数的值发生改变，可能语义发生改变等。此类问题通常对软件整体运行影响不大，但在某些特定场景下可能会引发问题。
4. **No（无风险）**：这种级别的变动是良性的，不会对依赖于该开源软件的其他软件造成负面影响。如字段不再被 final 修饰，由常量变为非常量；新增了一个常数等。这些变动一般不会对软件的兼容性产生不良影响。

每个级别反映了不同类型的 API 变动可能给依赖该项目的其他软件带来的潜在风险。在进行版本升级或者选择不同的依赖版本时，应当特别注意这些兼容性问题及其严重程度。

2 第二题

2.1 题目

版本间兼容性可信度量公式什么？等级划分表是什么？

2.2 解答

2.2.1 版本间兼容性可信度量模型的灵感来源

版本间兼容性可信度量模型的灵感来源于玻尔兹曼熵公式，这是一种在统计力学中用来描述系统微观状态数目与宏观状态之间关系的方法。在软件工程领域，这个概念被巧妙地借用过来用于衡量开源软件不同版本间的兼容性。

在该场景下，不同严重程度的兼容性问题数量（如 High、Medium、Low 和 No）被视为反映软件意外行为风险的“微观物理量”，而软件版本间的整体兼容性水平则类似于一个“宏观物理量”。通过这种类比，我们可以用风险的微观表现来表征开源软件版本间兼容性可信度的宏观特性。

2.2.2 公式推导及含义

为了计算版本间兼容性可信度，我们引入了玻尔兹曼加权熵 H 的概念，其计算方式如下：

$$H = \sum_{i \in \{h,m,l,n\}} \alpha_i \cdot \log(n_i + 1)$$

其中：

- n_h, n_m, n_l, n_o 分别表示检测出的严重程度为 High、Medium、Low、No 的兼容性问题数量；
- $\alpha_h, \alpha_m, \alpha_l, \alpha_o$ 分别是 High、Medium、Low、No 风险因子的归一化权重，这些权重反映了不同严重等级对整体不兼容性的贡献。

这里，$n_i + 1$ 的设置是为了确保当某类兼容性问题的数量为 0 时，其对应的熵不会导致未定义的情况（即对数函数的输入不能为 0）。同时，这也意味着当某类兼容性问题不存在时，它对该版本组合的不兼容性熵的贡献为 0。

然后根据玻尔兹曼加权熵 H，我们可以进一步计算可信值 K：

$$K = e^{-H}$$

其中 $H \in [0, +\infty)$，H 值越大，说明软件两版本间不兼容度越大，两者越不兼容；$K \in (0, 1]$，K 值越大，说明软件两版本间兼容性可信性越高，两版本间越兼容。为了将最终可信值 T 范围映射到 $[1, 10]$，进一步改进得到最终的软件版本间兼容性可信度量模型：

$$T = \begin{cases} 10e^{-H} & (0.1 \leq e^{-H} \leq 1) \\ 1 & (0 \leq e^{-H} < 0.1) \end{cases}$$

此公式将玻尔兹曼加权熵映射到一个更直观的尺度上，使得 T 值越接近 10，表示两个版本间的兼容性越高；反之，T 值越低，则表明两版本之间的兼容性越差。

2.2.3 等级划分表

在计算得到开源软件版本间兼容性可信值后，我们还要进行**开源软件版本间兼容性可信等级划分**：

- 在进行可信等级划分时，我们不仅考虑开源软件版本间兼容性可信值，还要考虑版本间 API 变更的规模（即 API 变更的数量），变更规模越大，不兼容度越高，可信度越低。
- 而 API 变更包括：良性变更（向下兼容的变更：严重程度为 No）和破坏性变更（不向下兼容的变更：严重程度为 High、Medium、Low），我们都会将其纳入考虑；
- 而另一方面，对于不兼容变更数量定义划分规则时的阈值是很难确定的，所以我们最终考虑使用不兼容变更密度作为可信等级划分的条件之一。

最终，我们得到**开源软件版本间兼容性可信等级划分表**：

等级	T	d_h	d_m	d_l	或满足
V 级	$9 \leq T$	$d_h = 0$	$d_m = 0$	$d_l \leq 0.4$	无
IV 级	$7 \leq T < 9$	$d_h = 0$	$d_m \leq 0.4$	$d_l \leq 0.7$	或 $9 \leq T$ 且不能评为 V 级别
III 级	$4 \leq T < 7$	$d_h \leq 0.4$	$d_m \leq 0.7$	-	或 $7 \leq T$ 且不能评为 IV 级别及以上
II 级	$2 \leq T < 4$	$d_h \leq 0.7$	-	-	或 $4 \leq T$ 且不能评为 III 级别及以上
I 级	$1 \leq T < 2$	-	-	-	或 $2 \leq T$ 且不能评为 II 级别及以上

其中，T 表示最终可信值，d_h，d_m，d_l 分别是 High、Medium、Low 的变更密度，计算公式如下所示：

$$d_i = \frac{n_i}{n_h + n_m + n_l + n_o}(i = h, m, l, o)$$

n_h，n_m，n_l，n_o 分别表示检测出的严重程度为 High、Medium、Low、No 的兼容性问题数量。

3 第三题

3.1 题目

阐述软件间兼容性测试框架以及面向 Java 和 C++ 的工具平台框架。

3.2 解答

3.2.1 软件间兼容性测试框架

软件间兼容性测试框架主要用于检测主软件与直接依赖的开源软件之间的兼容性，包含依赖库调用分析、API 差异分析和软件间兼容性分析三个主要部分，具体内容如下：

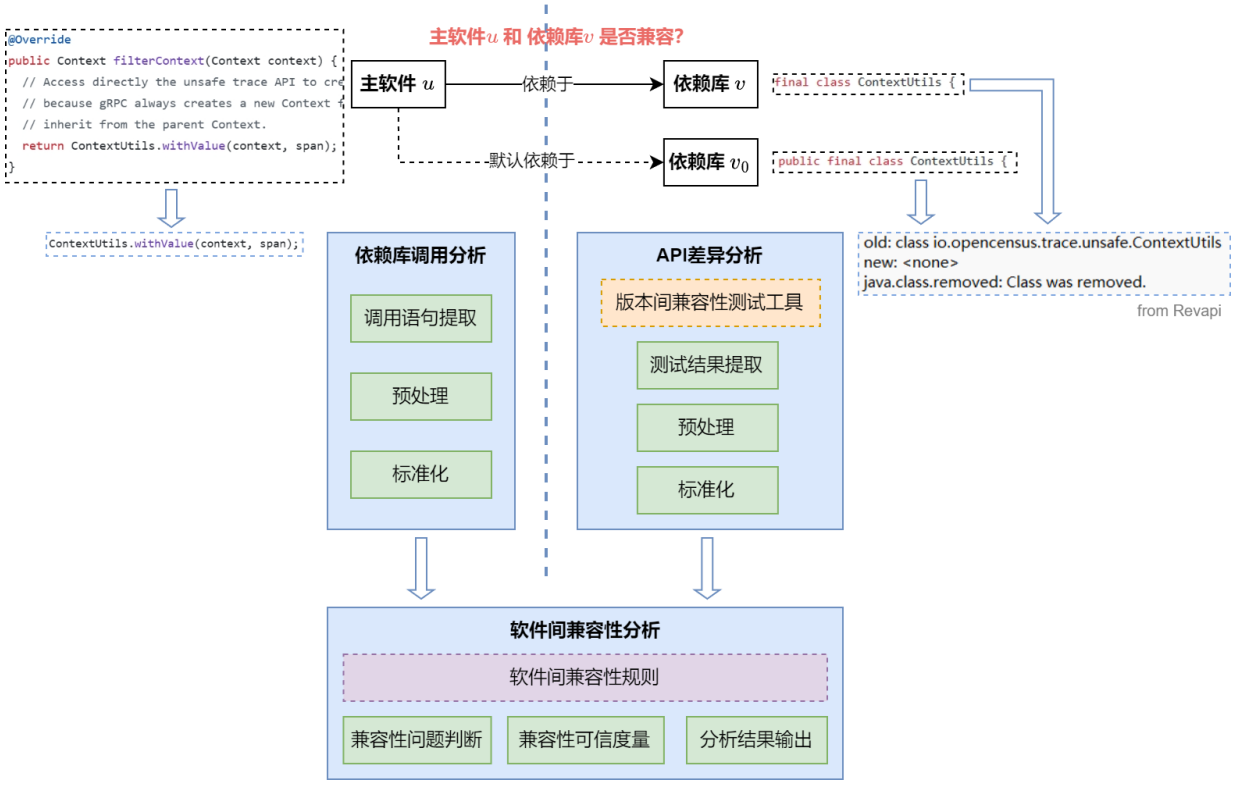

图 1: 软件间兼容性测试框架

1. 依赖库调用分析：

分析主软件当前版本的源代码，从中提取主软件对直接依赖软件中 API 调用语句，从而知道有哪些 API（类/接口/方法/…）被以何种形式调用（类继承/方法调用/方法重载…）。

2. **API 差异分析：**

   分析直接依赖软件的当前版本和默认版本中 API 存在的差异，即 API 相对于默认版本存在哪些变动（类被移除/方法参数数量变动/方法返回值类型变动…）。该分析可以借助已有的开源版本间兼容性测试工具进行。

3. **软件间兼容性分析：**

   针对从主软件源代码中的提取的 API 调用语句和直接依赖软件中的 API 差异，我们要分析这些 API 差异对 API 调用造成的影响，因为主软件通常只会调用直接依赖软件提供的部分 API。

   我们基于预先设定的软件间兼容性规则，来判断是否会存在兼容性问题，以及兼容性问题的严重程度。

值得一提的是，类继承经常引发兼容性问题。当直接依赖软件的某个抽象类中添加了新的抽象方法时，只有主软件继承了该类才会出现兼容性问题，如下图所示：

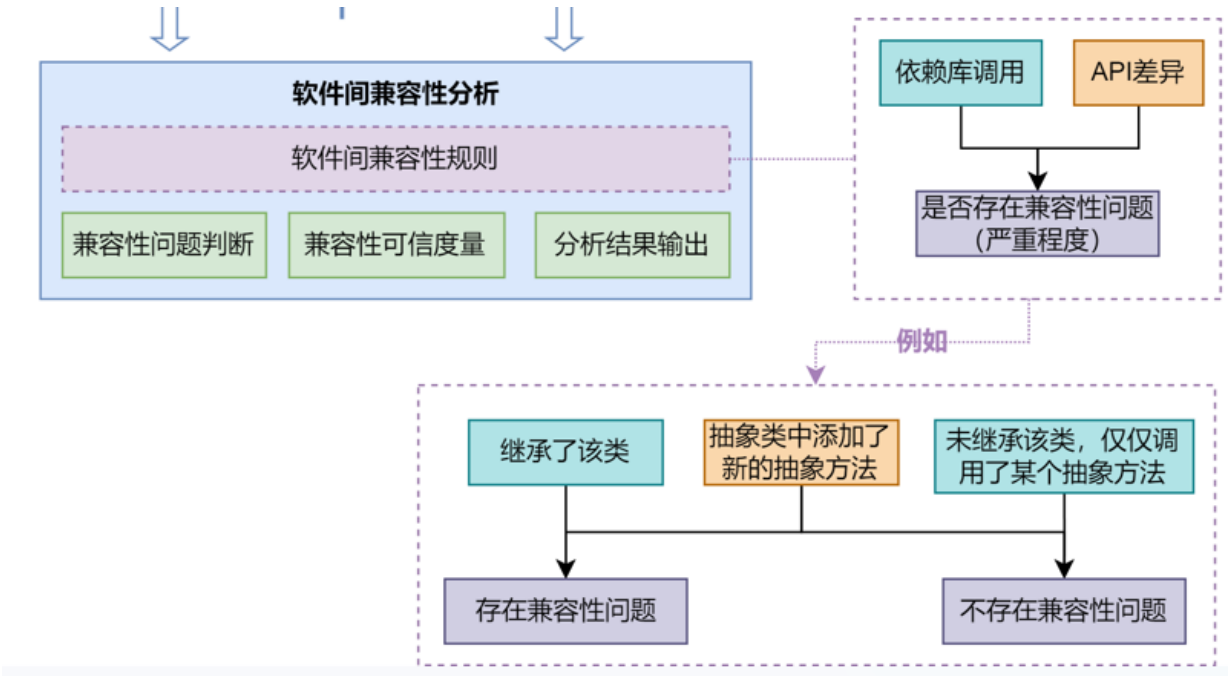

图 2: 类继承引发的兼容性问题

3.2.2 Java-Compatibility——面向 Java 的工具平台框架

Java-Compatibility 工具平台框架是专门设计用于检测和度量 Java 主软件与其直接依赖的开源软件之间的兼容性的。该框架基于前述软件间兼容性测试框架构建，它整合了静态分析与 API 差异分析技术，以确保能够准确地评估主软件在引入或升级其依赖项时可能遇到的兼容性问题。以下是 Java-Compatibility 工具的主要组成部分及其实现细节：

1. **依赖库调用分析**

在这个阶段，Java-Compatibility 使用 JavaParser 框架来解析主软件项目的源代码。通过静态分析，它可以识别出哪些 API（类、接口、方法等）被主软件所调用，并提取这些 API 调用语句。这一步骤至关重要，因为它确定了哪些 API 变动可能会对主软件产生影响。

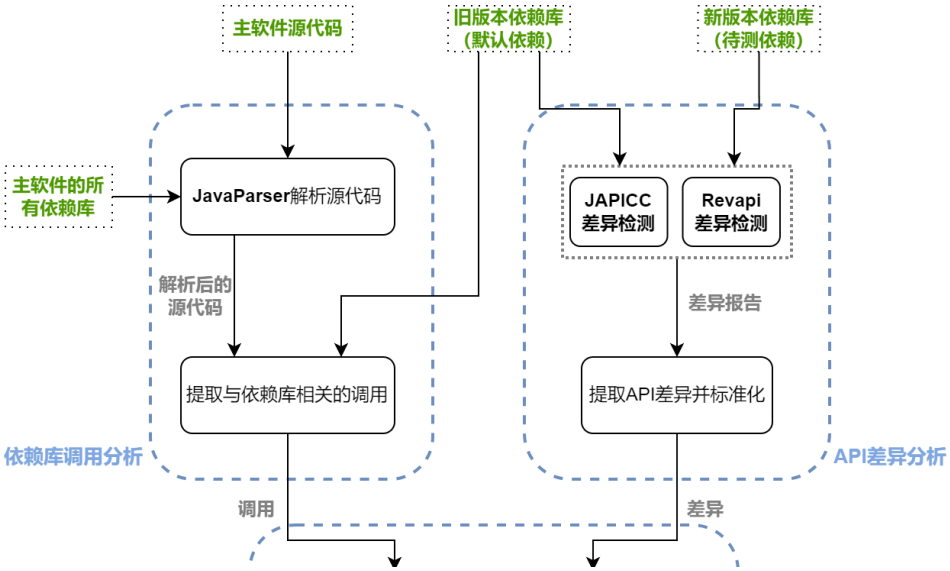

图 3: 依赖库调用分析、API 差异分析

2. **API 差异分析**

接着，Java-Compatibility 会利用 JAPICC 和 Revapi 工具来检测直接依赖软件当前版本与默认版本间的 API 差异。这两个工具可以生成详细的差异报告，指出 API 签名上的任何变化，如类或方法的添加、移除或修改。此外，Java-Compatibility 还会读取并处理这些差异报告，将它们标准化以便后续分析。

3. **软件间兼容性分析**

最后，在拥有主软件的 API 调用信息以及直接依赖软件的 API 差异之后，Java-Compatibility 会对两者进行综合分析。它基于预先设定的一套软件间兼容性规则，判断 API 变动是否会导致实际调用中的兼容性问题，并评估这些问题的严重程度。例如，如果一个抽象类中新增了一个抽象方法，并且主软件继承了这个类，则认为存在严重的兼容性问题。

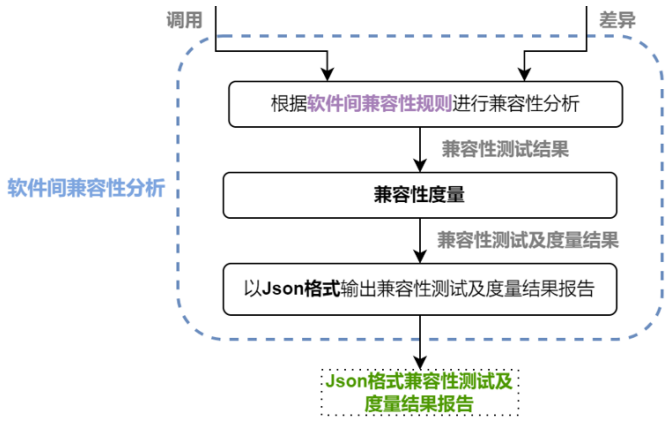

图 4: 软件间兼容性分析

Java-Compatibility 还实现了共 42 条针对 Java 语言的软件间兼容性规则。这些规则是在考虑 API 差异的基础上制定的，并结合具体的 API 调用情况。当一个 API 调用受到多个 API 差异的影响时，受到影响的严重程度总是取较高的值（即优先级为 High > Medium > Low > No）。下面是一些例子：

API 差异	API 差异描述	API 调用	严重程度等级
Class_Or_Interface_Removed	移除类/接口	调用了该类/接口	High
Method_Return_Type_Changed	方法返回值类型变动	该方法被调用	Medium

完成上述步骤后，Java-Compatibility 会计算最终的兼容性测试结果，并输出一份 JSON 格式的结果报告。这份报告不仅包含了检测到的所有兼容性问题及其严重程度，还包括了根据玻尔兹曼熵公式计算得出的软件间兼容性可信度量值 T。此值反映了两个软件之间兼容性的可靠程度，范围从 1 到 10 不等，数值越大表示兼容性越好。同时，还提供了基于不同严重程度兼容性问题密度的等级划分，帮助用户更直观地理解兼容性状况。

3.2.3 Cpp-Compatibility——面向 C++ 的工具平台框架

Cpp-Compatibility 工具平台框架是专门为检测和度量 C++ 主软件与其直接依赖的开源软件之间的兼容性而设计的。该框架同样基于前述软件间兼容性测试框架构建，整合了静态分析与 API 差异分析技术，确保能够准确评估主软件在引入或升级其依赖项时可能遇到的兼容性问题。以下是 Cpp-Compatibility 工具的主要组成部分及其实现细节：

1. **依赖库调用分析**

   在这个阶段，Cpp-Compatibility 将主软件源代码与其所有依赖库一同编译（在 Linux 环境下），并保留调试信息以生成动态链接库（.so）。通过使用 pyelftools 库读取和分析这些动态链接库中的调试信息，可以提取出主软件对依赖库的 API 调用情况。这一步骤对于确定哪些 API 变动可能影响到主软件至关重要。

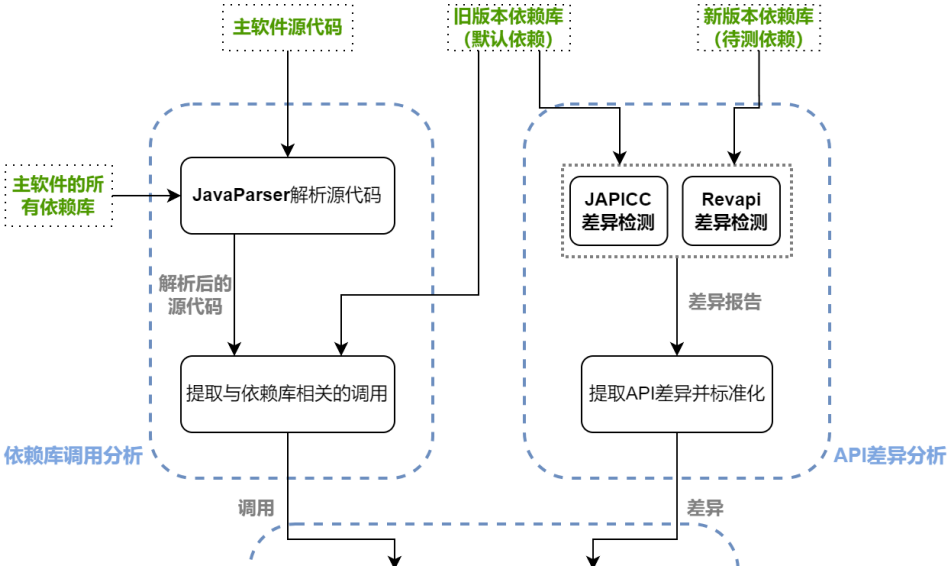

图 5: 依赖库调用分析、API 差异分析

2. **API 差异分析**

接着，Cpp-Compatibility 会利用 ABICC 工具来检测直接依赖软件当前版本与默认版本间的 API 差异。ABICC 工具能够生成详细的差异报告，指出 API 签名上的任何变化，如函数、数据类型等的添加、移除或修改。此外，Cpp-Compatibility 还会读取并处理这些差异报告，将它们标准化以便后续分析。

3. **软件间兼容性分析**

最后，在拥有主软件的 API 调用信息以及直接依赖软件的 API 差异之后，Cpp-Compatibility 会对两者进行综合分析。它基于预先设定的一套 C/C++ 软件间兼容性规则，判断 API 变动是否会导致实际调用中的兼容性问题，并评估这些问题的严重程度。例如，如果一个类中新增了一个纯虚方法，并且主软件继承了这个类，则认为存在严重的兼容性问题。

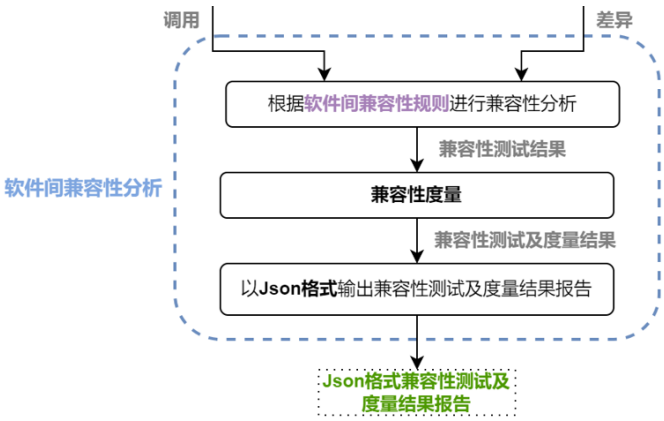

图 6: 软件间兼容性分析

Cpp-Compatibility 还实现了共 66 条针对 C/C++ 语言的软件间兼容性规则。这些规则是在考虑 API 差异的基础上制定的，并结合具体的 API 调用情况。当一个 API 调用受到多个 API 差异的影响时，受到影响的严重程度总是取较高的值（即优先级为 High > Medium > Low > No）。下面是一些例子：

API 差异	API 差异描述	语言限定	API 调用	严重程度等级
Removed_Field	数据类型中删除字段	C/C++	该字段被调用	High
Added_Pure_Virtual_Method	类新增纯虚方法	C++	该类被实例化	High
Field_Pointer_Level_Changed	字段指针等级改变	C/C++	该字段被访问	Medium
Changed_Constant	常数值发生改变	C/C++	该常数被引用	Low

完成上述步骤后，Cpp-Compatibility 会计算最终的兼容性测试结果，并输出一份 JSON 格式的结果报告。这份报告不仅包含了检测到的所有兼容性问题及其严重程度，还包括了根据玻尔兹曼熵公式计算得出的软件间兼容性可信度量值 T。此值反映了两个软件之间兼容性的可靠程度，范围从 1 到 10 不等，数值越大表示兼容性越好。同时，还提供了基于不同严重程度兼容性问题密度的等级划分，帮助用户更直观地理解兼容性状况。

此外，Cpp-Compatibility 的设计也充分考虑了 C++ 特有的特性，比如虚函数表、名称修饰（name mangling）等，以确保对 C++ 项目的兼容性分析更加精准和全面。具体来说：

- **虚函数表（Vtable）**：C++ 支持多态性，其中虚函数表用于实现运行时的多态调用。Cpp-Compatibility 能够识别虚函数的变化，包括虚函数的添加、删除或重载，这对于保证子类正确覆盖父类的方法非常重要。
- **名称修饰（Name Mangling）**：C++ 编译器为了区分不同作用域内的同名函数，会在编译过程中对函数名进行修饰。Cpp-Compatibility 能够解析这种修饰后的符号，从而准确跟踪 API 签名的变化，即使原始函数名保持不变。
- **模板（Templates）**：C++ 中的模板允许创建泛型类和函数。Cpp-Compatibility 特别关注模板定义和实例化的变化，因为这些变化可能会导致编译错误或者行为的不一致。
- **异常规范（Exception Specifications）**：C++ 允许函数声明抛出特定类型的异常。Cpp-Compatibility 会检查异常规格的变化，因为这些变化可能影响到程序的异常处理逻辑。
- **内联函数（Inline Functions）**：虽然内联函数主要是一个优化选项，但它们也可能影响二进制兼容性。Cpp-Compatibility 会注意内联函数定义的变化，尤其是当这些变化可能导致不同的行为时。
- **宏定义（Macros）**：预处理器宏定义在 C++ 中广泛使用。Cpp-Compatibility 会留意宏定义的变化，因为它们可以在编译时改变代码的行为，进而影响到依赖库的兼容性。

总而言之，Cpp-Compatibility 确保了对 C++ 项目兼容性分析的深度和广度，使得开发者能够在引入新的依赖版本时更有信心地评估潜在的风险。