ISSN 1000-9825, CODEN RUXUEW
*Journal of Software*, Vol.20, No.2, February 2009, pp.256−270
doi: 10.3724/SP.J.1001.2009.00561

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

# 基于客户程序度量包内聚性*

周天琳[1+], 徐宝文[1,2,3], 史 亮[4], 周毓明[2,3]

[1](东南大学 计算机科学与工程学院,江苏 南京 210096)

[2](南京大学 计算机软件新技术国家重点实验室,江苏 南京 210093)

[3](南京大学 计算机科学与技术系,江苏 南京 210093)

[4](微软中国研发集团,北京 100190)

## Measuring Package Cohesion Based on Client Usages

ZHOU Tian-Lin[1+], XU Bao-Wen[1,2,3], SHI Liang[4], ZHOU Yu-Ming[2,3]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[2](State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China)

[3](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[4](Microsoft China Research & Develop Group, Beijing 100190, China)

+ Corresponding author: E-mail: zhoutianlin@seu.edu.cn

**Zhou TL, Xu BW, Shi L, Zhou YM. Measuring package cohesion based on client usages. *Journal of Software*, 2009,20(2):256−270.** http://www.jos.org.cn/1000-9825/561.htm

**Abstract**: A number of package cohesion metrics have been proposed in the last decade, but they mainly converge on intra-package data dependencies between classes, which are inadequate to represent the semantics of packages in many cases. To address this problem, the authors first classify packages into four categories in terms of the kinds of their tasks. Next, a new package cohesion called CRC based on client usages is proposed by considering the fact that several classes are closely related if they are always reused together. And then the application areas of CRC in terms of the package classification framework are analyzed. Finally, a CRC measure called HC is presented. Compared to existing package cohesion metrics, HC considers not only intra-package but also inter-package data dependencies. It is hence able to reveal semantic relationships between classes. Furthermore, HC takes into account how the clients of a package use the package, thereby providing a finer-grain evaluation of the cohesion of a package. Experimental results demonstrates the effectiveness of HC, which likewise proves the feasibility of CRC.

**Key words**: software measurement; cohesion; package

**摘 要**: 为了一致而高效地计算包内聚性,许多研究者提出了大量的包内聚性度量方法.然而,这些方法主要依赖于包内部的数据流关系,常导致度量结果与实际开发经验相悖.为了解决这一问题,首先以包的职责为基础将包划分为 4 类.然后,提出了共同重用内聚 CRC,并根据包的分类框架讨论了 CRC 的适用性.CRC 的核心思想是若多个

* Supported by the National Science Foundation for Distinguished Young Scholars of China under Grant No.60425206 (国家杰出青年科学基金); the National Natural Science Foundation of China under Grant Nos.90818027, 60503033, 60633010 (国家自然科学基金); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2006094 (江苏省自然科学基金)

Received 2008-02-27; Accepted 2008-04-16

类总被共同重用,则它们之间存在紧密耦合.最后,提出了度量 CRC 的海明内聚度 HC.与现有方法相比,HC 同时考虑了包内和包间的数据依赖.因而,该方法能够有效地反映包内部类间的语义关系.此外,HC 利用包的使用模式提高了度量结果的可区分性.实验研究表明 HC 能够有效评估包的内聚程度.充分说明了作为 HC 基础的 CRC 具有较高的合理性.

**关键词:** 软件度量;内聚性;包

**中图法分类号:** TP311 **文献标识码:** A

## 1 Introduction

Packages enable engineers to reason about the design at a higher-level abstraction than classes, which have a vital role to develop and maintain large-scale software systems. In object-oriented systems, a *package* is a set of classes for providing a larger container to organize codes, generally corresponding to a directory. Its quality affects team development efficiency. Packages reflect the high-level structure of softwares, and thus determine the organization of the team to a very large degree[1,2]. Reasonable package organization provides well-defined interfaces and allows engineers to work parallelly. Besides, reasonable package organization helps to improve the understandability of softwares. Generally speaking, it is much easier for engineers to grasp the software architecture by quickly browsing the package structure rather than reading the codes. Finally, packages are the granules of release and reuse, which is significantly important for the maintainability, reusability, and extensibility of softwares[1−3].

As a quality indicator, *package cohesion* reflects how tightly-related the internal classes of a package are to contribute to a single task. Cohesion, originating in structured design, is a design attribute that can be used to predict properties of implementations[4]. It is generally accepted that "*the higher the cohesion of a module is, the easier the module is to develop, maintain, and reuse*"[5]. Many studies have provided empirical evidences for this wisdom[6−10]. In the context of package design, high cohesion of a package indicates that the classes in the package have close relations among themselves while performing a single task. This lowers the complexity of the package. Thus, the efficiency of the development and maintenance is improved. Moreover, from the viewpoint of clients, the clients of a cohesive package are not disturbed by the changes to the classes that they do not care about. Finally, the classes in a cohesive package are likely to be closed together against the same kinds of changes, which make "*the changes focus into a single package rather than have to dig through a whole bunch of packages and change them all*" [1]. It is hence easier for the evolution of softwares. We have demonstrated the importance of the high cohesion principle in guiding package refactoring and evolving[11].

A number of metrics have been proposed for evaluating module cohesion. However, there are some limitations in package cohesion measurement. Most existing cohesion metrics for packages only consider intra-package data dependencies. Many packages are still cohesive although they have few intra-package data dependencies, such as the package *AbstractTransaction* and *TransactionImplementation* in Payroll system[1] and *STL** in the C++ standard library. For such packages, their classes are coupled by semantics rather than dataflow. Thus, existing package cohesion metrics are unable to assess the cohesion of a package in many cases.

To address the above problem, we propose a new package cohesion called CRC (common reuse cohesion). On the assumption that several classes are closely related if they are always reused together, CRC makes use of client usages to evaluate the cohesion of a package. In this paper, we present a package classification framework and

---

The standard template library (STL) is a C++ standard library of container classes, algorithms, and iterators, which provides many of the basic algorithms and data structures of computer science, such as vector, find, sort, etc. See details at http://www.sgi.com/tech/stl/.

discuss the application areas of CRC based on the framework. Then, we define a package cohesion measure called HC (hamming cohesiveness) for CRC. Compared to existing works, we use the client behaviors of a package to infer the cohesion of the package, which involves both inter- and intra- package data dependencies. Moreover, we consider how the clients of a package use the package. In this way, we are able to gain better measurement results, thereby helping engineers evaluate package quality effectively and promoting the widespread use of package cohesion in the software development process.

The remainder of this paper is structured as follows. Section 2 describes the classification framework of packages and related works. Section 3 presents CRC and discusses its application areas, and then defines a CRC measure called HC. Section 4 illuminates the properties of HC. Section 5 demonstrates the effectiveness of HC by case studies. Section 6 concludes the paper and outlines directions for future work.

## 2 Background

As softwares increase in size and complexity, they need to be organized through larger units than classes. In the practical development, the larger units are generally packages, which allow engineers to think about the design at a higher level of abstraction. Thus, the quality of packages is important for large-scale softwares. As a design attribute, cohesion is widely used to evaluate the package quality. In this section, to investigate package cohesion we first classify packages into four categories according to the tasks that they perform. Then, we discuss the limitations of existing package cohesion metrics in terms of this package classification framework.

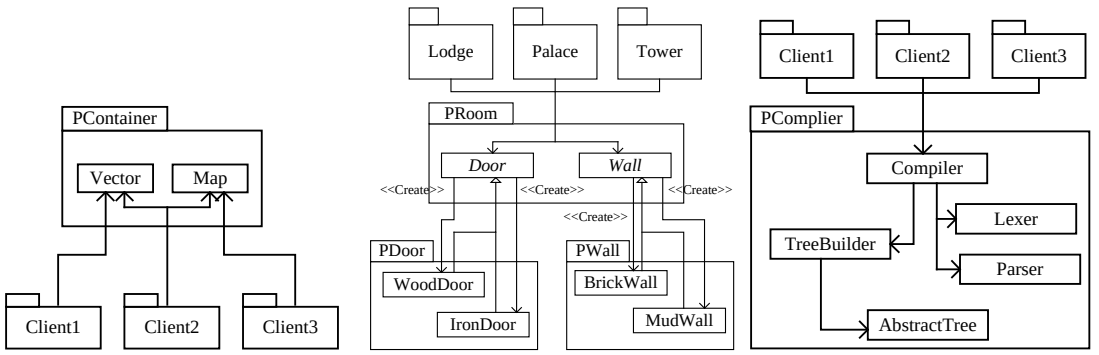
### 2.1 Package classification framework

The task of a package determines whether the package is cohesive or not[1,4,12,13]. A cohesive package usually performs a single task, whereas a non-cohesive package tries to carry out several ones. To investigate package cohesion, we classify packages into the following four categories according to the kinds of their tasks.

- A *utility package* is composed of basic classes that can be widely used in software development to support general programming tasks. This kind of packages usually exists in libraries, such as *STL* and *Boost* . A utility package *PContainer* consisting of collection classes (*Vector* and *Map*) is shown in Fig.1(a), where rectangles represent classes and packages and directed lines denote data dependencies. This package is designed for the prevalent requirements of managing a group of objects in software development.
- An *interface package* consists of abstract classes that express the specifications of a system. It insulates the affect of the changes of implementation details from clients, which conforms to Dependency Inversion Principle[1]. In Fig.1(b), the package *PRoom* contains two abstract classes Door and *Wall*, which represents the basic elements that a construction needs. Besides, *PRoom* makes the change of implementation details (the subclasses of *Door* and *Wall*) independent of its clients (the package *Lodge*, *Palace*, and *Tower*).
- An *implementation package*, the implementation of a system's specifications, is made up of subclasses inherited from abstract classes encapsulated in the same interface package. For instance, in Fig.1(b), the package *PDoor* and *PWall* respectively consist of implementation details of the abstract classes (*Door* and *Room*) in the package *PRoom*.
- A *component package* encapsulates a subsystem to accomplish a relatively complete task. As in Fig.1(c), the package *PCompiler* provides a subsystem for compiling source code, whose implementation uses Facade

Boost is a repository of free peer-reviewed portable C++ source libraries. See details at http://www.boost.org/.

pattern[14]. The class *Compiler* plays the role of Façade, and the other classes act as subsystem classes.




(a) Utility package (b) Interface & implementation package (c) Component package

Fig.1 Different kinds of packages

This package classification framework indicates that intra-package data dependencies are not enough to express the semantics of packages, because packages usually have few such dependencies except component packages. For utility, interface and implementation packages, the classes in the same package are usually independent of each other. Each class in a utility package often provides a small and independent functionality such as the class *Vector* and *Map* in Fig.1(a). The abstract classes in an interface package model independent concepts in a domain. Consider the package *PRoom* in Fig.1(b), the class *Door* and *Wall* respectively represent doors and walls in the domain of construction. And the classes in an implementation package are different implementations of a set of related abstract classes. As in Fig.1(b), the class *WoodDoor* and *IronDoor* provide two implementations of the abstract class *Door*. Unlike the former three kind packages, the classes in a component package often need to explicitly incorporate with other classes in the same package to achieve a relatively complete task such as the package *PCompiler* in Fig.1(c).

### 2.2 Related works

Cohesion, as an important design attribute, is closely related to the cost of debugging, maintenance, and modification. A great deal of experimental studies have shown that cohesion can be used to predict the external attributes of softwares, such as the fault-proneness of classes[6–8], understandability[9], etc. Almost every engineer thus strives to improve cohesion. However, it is not easy for engineers to conduct module cohesion assessment by manual review[15], especially in large-scale softwares. To solve this, researchers have proposed many cohesion metrics. The major researches focus on classes[5,8,16–22]. We have also explored this domain in depth. We have proposed several class cohesion metrics[23–27], and reviewed on some typical metrics[28–30]. Besides, we have explored the importance of class cohesion in predicting class faults by empirical analysis[10].

In the domain of package cohesion measurement, many studies are proposed for Ada. Patel et al. extended the concept of document similarity in information retrieval and measure the similarity between subprograms using the shared data types. And the cohesion of a package is computed to be the average of the similarity measures over distinct pairs of subprograms[31]. Briand et al. defined *cohesive interaction graph* to represent the design-level interactions between elements and proposed a cohesion metric suite based on the graph[6]. And we discussed the relationships among the entities of a package in terms of dependency analysis and then utilized the dependencies to define package cohesion measures[32]. However, in Ada, a package is used to represent a logical grouping of declarations that can be imported into other programs. It plays the similar role as class in other languages such as Java and C++. It is hard for these metrics to incarnate that packages are larger units than classes. Therefore, in this paper, we place the cohesion metrics for Ada packages into the category of the metrics for classes.

Besides cohesion metrics for Ada packages, Martin proposed RC (relational cohesion). RC is defined as the ratio of the number of relations in a package to the number of classes in the package[1]. However, the maximal value of RC varies with the number of classes in a package, which violates the properties that a good cohesion measure should satisfy. According to Briand, *et al*., a good module cohesion measure should have minimum and maximum[16,33].

Allen *et al*. proposed an information-theory-based metric to evaluate modular cohesion. They first use a graph to represent the relations between elements in a module. Then, they regard such a graph as an information source and model the pattern of edges incident to each node as a random variable. Finally, module cohesion is defined as excess entropy (EEC)[34]. Since EEC only requires *intramodule-edges graph*, it is easy to apply it to package.

However, both RC and EEC are only based on intra-package data dependencies between classes, thereby not being adapted to utility, interface and implementation packages. According to Section 2.1, except component packages, utility, interface and implementation packages have few intra-package data dependencies. Therefore, RC and EEC are only appropriate to component packages. For example, consider the packages shown in Fig.1, RC and EEC just make sense for the package *PCompiler*. And owing to no intra-package data dependencies, the RCs and EECs of the other packages equal to 0 in spite of their high semantic cohesion.

Although CPC (contextual package cohesion) proposed by Ponisio tries to expose implicit dependencies between classes in the view of clients, the formula of CPC is inconsistent with the definition of CPC[35]. In the formula, the clients of a package are defined as classes. But in the definition, the clients are packages. This results in different measurement results in some cases. So we do not discuss this method in this paper.

As mentioned above, the existing cohesion metrics are mainly designed for component packages. Cohesion, in its nature, is determined by semantics. The existing metrics have difficulties to understand the semantics of a package, and only rely on intra-package data dependencies to speculate whether the package performs a single task. They suppose that the closer the intra-package data dependencies are, the more cohesive the package is. However in practice, the close data dependencies in a package primarily indicate that its classes are involved in a common calculation, not a common task. This hence induces the invalidation of these metrics in many cases. Therefore, such kind of low-level relations (intra-package data dependencies) are insufficient to evaluate the cohesion of a package. In the rest of this paper, we say “dependency” as a short for “data dependency” for convenience.

We believe that client usages are able to reflect the semantics of packages such that they can be used to evaluate the cohesion of packages. In real world, engineers always design or comprehend a package by means of the application contexts in which the package exists. As crucial part of the application contexts, the client usages have capability of expressing the semantics of the package. In the domain of web site design, we have utilized users’ surfing actions to measure web site navigability by Markov model[36]. It is therefore similarly feasible to investigate the cohesion of a package in the view of client usages. In the next section, we will discuss how to use client usages to evaluate package cohesion.

## 3 Common Reuse Based Package Cohesion Measure

In this section, we first propose a new kind of package cohesion called CRC, which is adapted to the semantics of packages. Then, we present a measure called HC to calculate CRC, which considers both inter- and intra-package dependencies.

### 3.1 Common reuse cohesion

The cohesion of a package is able to be measured in the view of client usages, namely how the clients use the package. Common reuse principle (CRP) states that the classes in a cohesive package are always reused together by

its clients[1]. From the viewpoint of semantics, CRP reflects that classes always reused together are coupled semantically. Generally speaking, it is seldom for a class to be reused solely. To achieve one task, a reusable class needs to collaborate with other classes that are part of the reusable abstraction, such as the class *Door* and *Wall* in Fig.1(b). Indeed, the classes in a cohesive package always show a consistent abstraction for the domain that the package serves to the clients of the package. From the angle of development practices, CRP can avoid unnecessary rerelease and revalidation. When an engineer uses a package, a dependency is generated upon the whole package. Since then, whether the engineer uses all the classes in the package or not, every time that package is changed, the clients must be rereleased and revalidated, even if the change is in a class that the clients do not care about[1].

In terms of CRP, a package is the most cohesive if all the classes in the package are reused together by all its clients. Contrarily, a package is the most non-cohesive if no classes are reused together. In the following, we provide the definition of CRC for assessing package cohesion.

**Definition 1** (**CRC**). CRC (common reuse cohesion) of a package is the degree that the classes in the package are commonly reused by the clients of the package: the more the classes are reused together, the higher the CRC is.

For example, the package *PCompiler* in Fig.1(c) is cohesive, since all its classes are reused together.

CRC has the following advantages. First of all, CRC reflects the relevance between the modification of a package and its clients. In general, the modifications of a package with high CRC are always relevant to all its clients, which guarantees that the clients are not disturbed by unrelated changes of the package. Contrarily, the clients of a package with low CRC are often influenced by unrelated changes. Moreover, CRC uses client usages to mine the semantic couplings between classes, thereby being suitable for evaluating package cohesion. And compared to other methods, the applicability of CRC is enhanced. In the following, we discuss in detail the applicability through the package classification framework given in Section 2.1.

- CRC may not be very effective for utility packages. In the practical development, engineers usually use only a part of classes in utility packages according to their requirement. For example, the package *PContainer* in Fig.1(a) has the low CRC, since the class *Vector* and *Map* are not reused together in most cases. But both the two classes are collections of objects in concept. And it is obvious that the CRCs of some other utility packages such as *STL* are also not very high. Nevertheless, it is feasible to observe package cohesion via client behaviors, which conforms to the way that engineers design or comprehend packages. We will hence conduct further study on the relationship between CRC and utility packages in the future works.
- Interface packages should be treated in accordance with different cases. If an interface package is in libraries, CRC may be inapplicable. The reason is similar to utility packages. If the package is in an application, such as the package *PRoom* in Fig.1(b), CRC usually works well.
- For implementation packages, the applicability of CRC is the same as that of interface packages. This is because implementation packages are the extension to the corresponding interface packages. When using an abstract class, it is nature to also use its concrete subclasses. For example in Fig.1(b), the package *Lodge*, *Palace* and *Tower*, in general, also use the subclass *WoodDoor*, *IronDoor*, *BrickWall* and *MudWall*.
- CRC is widely applicable to component packages. For well-designed component packages, every class in them should be reused directly or indirectly. Consider the package *PCompiler* in Fig.1(c), the class *Compiler* is reused directly, while the class *Lexer*, *Paser*, *TreeBuilder* and *AbstractTree* are reused indirectly.

In summary, CRC is effective to measure the quality of applications. However, for packages in libraries, further research need be done to study the effectiveness of CRC by gathering a mass of clients from open source projects. Since most engineers write codes for applications, we restrict ourselves to the context of developing applications in this paper.

### 3.2 Measure definition

To properly evaluate CRC of a package, we should first identify the dependencies between classes. Generally, classes have two basic dependencies: inheritance and usage. A class $c_1$ has an *inheritance dependency* on another class $c_2$, denoted by $c_1 \xrightarrow{i} c_2$, if $c_2$ is an ancestor class of $c_1$. And a class $c_1$ has a *usage dependency* on another class $c_2$, denoted by $c_1 \xrightarrow{u} c_2$, if a method implemented in $c_1$ references a method or an attribute implemented in $c_2$. Here, an inherited method is considered as a method of another class. Besides, due to dynamic binding, we can not statically determine which method would be invoked at runtime. And thus, a conservative way is employed, which takes all the methods that can be bound into account. For example, in Fig.1(b), the class *WoodDoor* and *IronDoor* have inheritance dependencies on the class *Door*, and the class *Door* has usage dependencies on the class *WoodDoor* and *IronDoor*.

**Definition 2** (**Class dependency**). A class $c_1$ has a dependency on another class $c_2$, denoted by $\longrightarrow$, if $c_1$ has an inheritance or usage dependency on $c_2$. A class $c_1$ has an indirect dependency on another class $c_2$, denoted by $c_1 \xrightarrow{+} c_2$, if $(c_1, c_2)$ belong to the transitive closure of $\longrightarrow$.

As mentioned above, the CRC of a package can be measured by computing the degree to which its classes are reused together by its clients. For accuracy, we give a formal definition of the clients of a package:

**Definition 3** (**Package client**). A package $p_1$ is a client of another package $p_2$, if there is a class $c_1$ in $p_1$ that has a usage dependency on a class $c_2$ in $p_2$. Let ClientPackage($p_1$) be the set of the clients of $p_1$.

In Definition 3, we exclude the implementation packages from the client set of their corresponding interface packages. As for an interface package, the task of its implementation packages is to implement the specification that it describes, not using its functions. Thus, we do not regard implementation packages as clients of their corresponding interface packages. Consider the package *PRoom* in Fig.1(b), ClientPackage(*PRoom*) = {*Lodge*, *Palace*, *Tower*}.

To evaluate whether the classes in a package are reused together, we need to identify how its clients use its classes. Vector is a good tool to express pattern in many domains. Hence, we utilize it to define use pattern of packages.

**Definition 4** (**Use pattern**). For a package $p_1 = \{c_1, c_2, \ldots, c_m\}$ and its client $p_2$, the *use pattern* of $p_2$, denoted by UsePattern($p_1$, $p_2$), is a vector $(v_1, v_2, \ldots, v_m)$, where $v_i \in \{0, 1\}$. Let $c_i \in p_1$, the value of corresponding component $v_i$ is determined in the following way:

$$v_i = \begin{cases} 1, & \text{if } \exists c \in p_2 \wedge c \xrightarrow{+} c_i \\ 0, & \text{if } \neg\exists c \in p_2 \wedge c \xrightarrow{+} c_i \end{cases}.$$

According to Definition 4, when UsePattern($p_1$, $p_2$) is equal to (1, 1,…,1), $p_2$ uses all the classes in $p_1$. Hence, we call (1,…,1) as *best pattern*, denoted by BestPattern($p_1$). For example in Fig.1(b), UsePattern(*PRoom*, *Lodge*) = UsePattern(*PRoom*, *Palace*) = UsePattern(*PRoom*, *Tower*) = BestPattern(*PRoom*) = (1, 1).

In terms of CRC, if a package $p$ is cohesive, then its clients use $p$ in a similar way. Furthermore, the similar way is to use all its classes. Then, the use patterns of $p$'s clients usually satisfy the following property:

$\forall p_i \in$ ClientPackage($p$), the similarity between UsePattern($p$, $p_i$) and BestPattern($p$) is high.

Herein, the similarity between use patterns can be evaluated by any similarity measurement employed in information retrieval. Consider the component of the use pattern belong to {0, 1}, we can use Hamming distance to compute the similarity. *Hamming distance* between two vectors $V_1$ and $V_2$ is simply defined as the number of corresponding components that differ[37], denoted by HamDist($V_1$, $V_2$). Then, the similarity between $V_1$ and $V_2$ is computed as follows:

$$HamSim(V_1, V_2) = \frac{m - HamDist(V_1, V_2)}{m}, \text{ where } m \text{ is the length of } V_1(V_2).$$

For example, let $V_1$ = (0, 1), $V_2$ = (1, 1), HamDist($V_1$, $V_2$) = 1, and HamSim($V_1$, $V_2$) = 1/2.

According to the above property, for a package, the average of the HamSims over its best pattern and the use patterns of its clients should reach a big value. Thus, the CRC measure, *Hamming Cohesiveness* (HC), is defined as the average HamSim over the best pattern and the use patterns of the clients.

**Definition 5** (**CRC measure**). For a package $p$, let m =|$p$|, $n$ = |ClientPackage($p$)|, we have

$$HC(p)=\begin{cases}\dfrac{\dfrac{m}{n}\displaystyle\sum_{p_i\in ClientPackage(p)} HamSim(UsePattern(p,p_i),BestPattern(p))-1}{m-1}, & \text{if } m>1\\ 1, & \text{if } m=1.\end{cases}$$

When $m$=1, there is only one class in $p$. In this case, $p$ has CRC obviously. We thus set HC($p$) to 1. When $m$ > 1, if each client of $p$ only uses one class in $p$, HC($p$) = 0. On the other hand, if each client uses all the classes in $p$, HC($p$) = 1. Thus, HC($p$)∈ [0, 1].

In Fig.1(b), for the package *PRoom* and the package $p_i$∈ClientPackage(*PRoom*), we have $m$=|*PRoom*|=2, $n$= |ClientPackage(*PRoom*)|=3, and HamSim(UsePattern(*PRoom*, $p_i$), BestPattern(*PRoom*))=1. According to Definition 5, HC(*PRoom*)=1.

## 4 Properties of HC

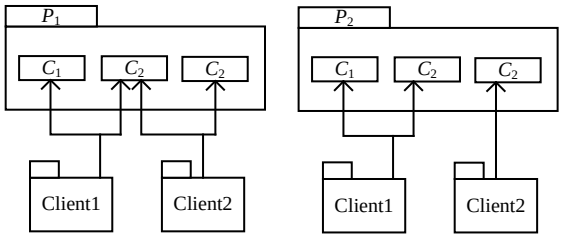
In terms of the definitions in Section 3, we can derive some properties of HC. Above all, HC measures package cohesion in the view of client usages. This enables HC to consider both inter- and intra- package dependencies. By inter-package dependencies, HC exposes the semantic couplings between classes through clients, which helps to evaluate package cohesion on a high abstract level. Indeed, the classes tending to be reused together always serve for a common abstraction regardless of whether there are dependencies between them, such as the class *WoodDoor* and *IronDoor* in Fig.1(b). Intra-package dependencies also contribute to package cohesion obviously, which help us allow for internal classes that are not directly used by clients. For example, in Fig.1(c), the class *Lexer*, *Paser*, *TreeBuilder*, and *AbstractTree* are considered via intra-package dependencies.

Secondly, HC is more effective when the clients employed in the computation of HC are sufficient. A sufficient set of the clients of a package can ensure that the measurement value is stable. In terms of the above discussion, the key idea of HC is to use the client behaviors to inspect the semantics of the package. Generally speaking, the more the use scenarios of the package are considered, the more the semantics of the package are mined. Therefore, the sufficient clients of the package can help HC to comprehend the complete semantics of the package. And in this case, the measurement value may not change much when some new clients are added to evaluate HC. This is always enough to assist engineers to judge whether the package is cohesive or not.

In the practical measurement, the sufficient set of the clients of a package should be collected in light of the situation that the package is reused. For a package designed for only one project, the term “sufficiency” means that all its clients should be considered. Likewise, for a package reused by several projects in a limited area such as in a company, all its clients should be also gathered for evaluating its HC. And for a package in wide use, it is very difficult, even infeasible, to collect all the clients since the clients are always scattering in enormous applications. Therefore, it suggests that a sufficient set of the clients should cover all the typical use scenarios of the package. Open source projects are generally good source for gaining sufficient and typical clients.

Thirdly, HC utilizes use pattern to reflect how clients of a package use the package. This allows finer-grain distinctions than counting. For example in Fig.2, if using the number of the clients of a package that use all its classes to evaluate its cohesion, then the measurement result of $P_1$ and $P_2$ are both 0. This is inconsistent with our experience. But

HC can distinguish the difference between the two packages: HC($P_1$) = 0.50 and HC($P_2$) = 0.25.




(a) (b)

Fig.2 Two cases that can not be distinguished by counting

Furthermore, HC satisfies the properties that a good module cohesion measure should satisfy[16,33].

1) non-negative and standardization
2) minimum and maximum
3) monotony
4) cohesion does not increase when combining two modules if there is no relationship between them.

According to the discussion in Section 3.2, HC∈[0, 1], therefore satisfying the property 1) and 2) clearly. Then, we will discuss whether HC satisfies the other two properties or not.

**Theorem 1**. For a package $p_1$, HC($p_1$) does not decrease when adding a dependency on a class in $p_1$.

*Proof*: Let $c_1 \in p_1$, when adding a dependency $c_2 \longrightarrow c_1$, we have:

If $c_2 \in p_2$, and $p_2 \in$ ClientPackage($p_1$), then $p_2$ will use the class $c_1$ and the classes in $p_1$ that $c_1$ depends on. According to Definition 4, more components of UsePattern($p_1$, $p_2$) will be 1. Therefore, HamSim(UsePattern($p_1$, $p_2$), BestPattern($p_1$)) become higher. Besides, for a package $p_3 \in$ ClientPackage($p_1$), and $p_3 \neq p_2$, HamSim(UsePattern($p_1$, $p_3$), BestPattern($p_1$)) remains the same. Therefore, HC($p_1$) does not decrease.

If $c_2 \in p_1$, then an arbitrary class $c_3$ that uses the class $c_2$ will also use the class $c_1$, where $c_3 \in p_2$ and $p_2 \in$ ClientPackage($p_1$). This is equivalent to adding a dependency from the class $c_3$ to the class $c_1$. Hence, HC($p_1$) does not decrease.

Consequently, HC($p_1$) does not decrease when adding a dependency on a class in $p_1$. □

**Theorem 2**. For the package $p_1$, $p_2$, and here are no relationships between $p_1$ and $p_2$, let $p_3 = p_1 \cup p_2$, then HC($p_3$) ≤Max(HC($p_1$), HC($p_2$)).

*Proof*. In the context of HC, there are no relationships between the two packages $p_1$ and $p_2$ if they satisfy both

$$\neg\exists c_1 \in p_1 \wedge \neg\exists c_2 \in p_2 : c_1 \longrightarrow c_2 \vee c_2 \longrightarrow c_1$$

and

$$\text{ClientPackage}(p_1) \cap \text{ClientPackage}(p_2) = \varnothing.$$

Consequently, when combining $p_1$ and $p_2$, the number of the classes in $p_3$ is equal to the sum of the number of the classes in $p_1$ and $p_2$. Besides, for every client of $p_1$ ($p_2$), it uses the same classes in $p_3$ with those in $p_1$ ($p_2$). Let $m_1$=|$p_1$|, $m_2$=|$p_2$|, $n_1$=|ClientPackage($p_1$)|, and $n_2$=|ClientPackage($p_2$)|, we have:

If $m_1$=1 or $m_2$=1, then HC($p_1$) or HC($p_2$) equals to 1. And HC is not greater than 1, thus, HC($p_3$)≤ Max(HC($p_1$), HC($p_2$)).

If $m_1 \neq 1$ and $m_2 \neq 1$, then,

$$HC(p_1) = \frac{\frac{m_1}{n_1} \sum_{p_i \in ClientPackage(p_1)} Similarity(UsePattern(p_1, p_i), BestPattern(p_1)) - 1}{m_1 - 1},$$

$$HC(p_2)=\frac{\frac{m_2}{n_2}\sum_{p_i\in ClientPackage(p_2)} Similarity(UsePattern(p_2,p_i),BestPattern(p_2))-1}{m_2-1}.$$

After combining, the length of the use patterns for $p_3$ is $m_1+m_2$. Because of no relationships between $p_1$ and $p_2$, for a package $p\in$ClientPackage($p_3$), UsePattern($p_3$, $p$) is determined by:

$$UsePattern(p_3,p)=\begin{cases}(UsePattern(p_1,p),0,...,0) & \text{if } p\in ClientPackage(p_1) \text{ and there are } m_2 \text{ zeros}\\ (0,...,0,UsePattern(p_2,p)) & \text{if } p\in ClientPackage(p_2) \text{ and there are } m_1 \text{ zeros}\end{cases}.$$

Therefore,

$$HC(p_3)=\frac{\frac{1}{n_1+n_2}((HC(p_1)\times(m_1-1)+1)\times n_1+(HC(p_2)\times(m_2-1)+1)\times n_2)-1}{m_1+m_2-1}.$$

Without loss of generality, assume that HC($p_1$)≤ HC($p_2$), $m_1\le m_2$, then we have

$$HC(p_3)-HC(p_2)\le\frac{HC(p_2)\times(m_2-1)}{m_1+m_2-1}-HC(p_2)=\frac{-HC(p_2)\times m_1}{m_1+m_2-1}\le 0.$$

Thus, HC($p_3$)≤Max(HC($p_1$), HC($p_2$)). □

In all, HC does not increase when combining two packages if there are no relationships between them. Therefore, HC satisfies Briand's properties.

## 5 Case Studies

In this section, we compare our approach with the existing package cohesion metrics via two cases. The first is performed on the packages in Fig.1, which is simple but contains all kinds of packages. Its simplicity excludes the trivial details in large-scale softwares, which makes us focus on the crucial problems. The second is the module *Evolution* in Evolutionary Testing Framework (ETF)[38–40]. ETF is an evolutionary testing framework developed by us in C++, which uses genetic algorithm to generate test data that can cover given sentences or paths of a program. As for ETF, *Evolution* is a relative independent subsystem to be charged with the task of evolutionary computing. *Evolution* is composed of 71 classes, and these classes are grouped into 15 packages. Figure 3 presents the main

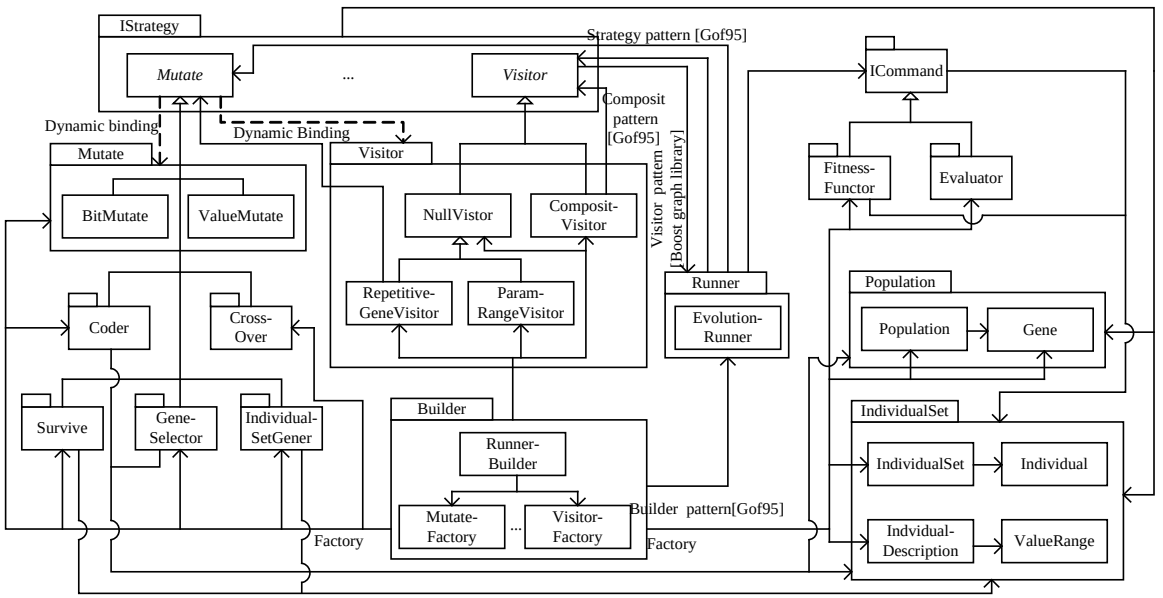



Fig.3 Main package structure of evolution

package structure of *Evolution*. In Fig.3, some important packages like the package *Visitor* are shown in detail. The detailed discussion about the cohesion of the packages in *Evolution* will be conducted in the following. Compared to the first case, *Evolution*, reserving the main characteristics of real-world softwares, is more like product codes. It allows us to observe the effectiveness of HC in the real world.

Tables 1 and 2 respectively show the RCs, EECs, HCs of the packages in Fig.1 and Evolution. And Figs.4, 5 and 6 respectively show the RCs, EECs, HCs of the interface, implementation, and component packages in both the cases. Since there is only one utility package in both cases, we do not draw a figure for it. The measurement results are gained by sufficient clients, so they are stable and meaningful. Although we just have conducted two case studies, the cases are complex enough for us to get some conclusions. Table 3 shows the comparison results of RC, EEC and HC.

**Table 1** Cohesion metrics of packages in Fig.1

| Package | RC | EEC | HC | Category |
|---|---|---|---|---|
| *PContainer* | 0.00 | 0.00 | 0.33 | Utility package |
| *PRoom* | 0.00 | 0.00 | 1.00 | Interface package |
| *PDoor* | 0.00 | 0.00 | 1.00 | Implementation |
| *PWall* | 0.00 | 0.00 | 1.00 | Package |
| *PCompiler* | 1.00 | 0.46 | 1.00 | Component package |

**Table 2** Cohesion metrics and descriptions of packages in evolution

| Packages | RC | EEC | HC | Descriptions | Category |
|---|---|---|---|---|---|
| *IStrategy* | 0.00 | 0.00 | 0.57 | Abstract evolution strategies | Interface package |
| *ICommand* | 0.00 | 0.00 | 0.70 | Controls the evolution process | |
| *Coder* | 0.00 | 0.00 | 1.00 | Concrete evolution strategies | Implementation package |
| *Mutate* | 0.00 | 0.00 | 1.00 | | |
| *Crossover* | 0.00 | 0.00 | 1.00 | | |
| *Survive* | 0.00 | 0.00 | 1.00 | | |
| *GeneSelector* | 0.00 | 0.00 | 1.00 | | |
| *IndividualSetGener* | 0.00 | 0.00 | 1.00 | | |
| *Visitor* | 0.75 | 0.23 | 1.00 | | |
| *FitnessFuntor* | 0.00 | 0.00 | 1.00 | Concrete command | |
| *Evaluator* | 0.00 | 0.00 | 1.00 | | |
| *Population* | 1.00 | 1.00 | 0.88 | Models population in evolution | Component package |
| *IndividualSet* | 0.50 | 0.33 | 0.74 | Models domain individuals | |
| *Builder* | 1.00 | 0.23 | 1.00 | Builds evolution runner | |
| *Runner* | 1.00 | 1.00 | 1.00 | General evolutionary process | |

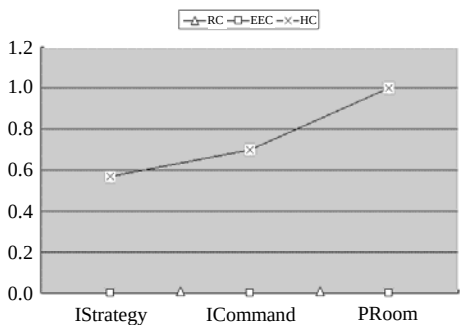



Fig.4 Measurement results of interface packages

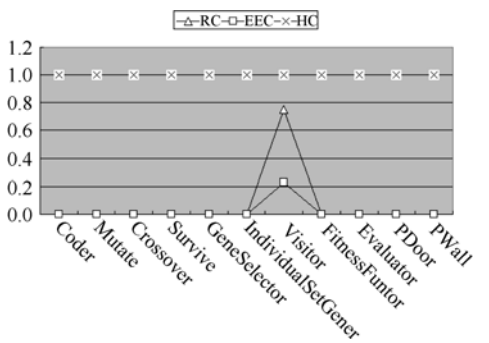



Fig.5 Measurement results of implementation packages

(1) For utility packages, RC and EEC are usually inapplicable, and HC may be not very effective.

In Table 1, we find that both RC(*PContainer*) and EEC(*PContainer*) are 0, which comes from few intra-package dependencies in utility packages. Besides, HC(*PContainer*) is just 0.33, a low value. This is because it is not unusual for engineers to use part of a utility package such as *STL*. However, *PContainer* is cohesive.

(2) For interface packages, RC and EEC may be unsuitable, while HC usually works well.

Most interface packages in the two cases are cohesive semantically except for the package *IStrategy*. The package *PRoom* in Fig.1 is cohesive in terms of Section 2.1. And the package *ICommand* consists of functors to control evolutionary process, such as fitness function.

However in Fig.4, all the RCs and EECs of the interface packages equal to 0, which indicates that they are often unfit for interface packages. The reason is that there are few intra-package dependencies in interface packages. And generally, an abstract class represents a relatively independent concept of the domain. Thus, there are always few dependencies between abstract classes.

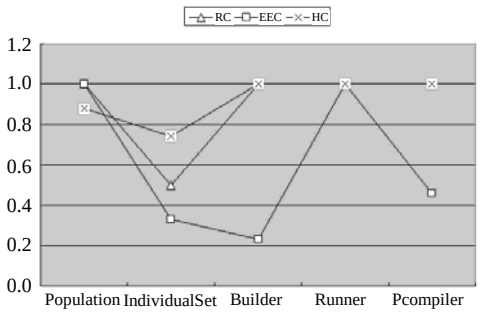



Fig.6 Measurement results of component packages

Also in Fig.4, the HCs for interface packages are much higher than the RCs and EECs. This shows that client usages are able to reveal semantic relations between abstract classes. Besides in Table 2, we note that HC(*IStrategy*) is only 0.57. The non-high value indicates that the package *IStrategy* that encapsulates evolutionary strategies such as mutate and coder is not very cohesive. From the viewpoint of semantics, unlike other classes such as *Mutate*, the class *Visitor* does not belong to the domain of evolutionary computing. Its function is to monitor the evolutionary process and collect data for our research. Therefore, the package *IStrategy* should be less cohesive than the package *PRoom* and *ICommand*. From the viewpoint of implementation, the package *Visitor* is not only the extension to the package *IStrategy*, but also a client. As shown in Fig.3, the class *CompositeVisitor* and *RepetiveGeneVisitor* in the package *Visitor* respectively have usage dependencies on the class *Visitor* and *Mutate* in the package *IStrategy*, and UsePattern(*IStrategy*, *Visitor*) = (1, 1, 0, 0, 0, 0, 0, 0). The use pattern that is far from BestPattern(*IStrategy*) lowers HC(*IStrategy*) sharply. Therefore, HC reflects the practical cohesion to a great degree.

(3) HC works effectively on implementation packages, whereas RC and EEC are usually unfit.

Implementation packages in Fig.1 and *Evolution* are cohesive, since every one is composed of the subclasses inherited from only one abstract class.

In Fig.5, all the RCs and EECs of the implementation packages are equal to 0 except for the package *Visitor*. This illustrates that the traditional metrics usually work ineffectively on implementation packages. The reason is that the subclasses of an abstract class usually do not have dependencies on each other, such as the class *WoodDoor* and *IronDoor* in Fig.1(b). Besides, we also find that all the HCs of the implementation packages are 1, which shows that HC are generally suitable for implementation packages. This is because that all the subclasses are reused together in the two cases. In our cases, there are two kinds of clients for the implementation packages. The first kind is factory/builder packages that obviously use all the classes in the implementation packages. The second kind is the clients of their corresponding interface packages. Because we adopt a conservative method to analyze dependencies, if a class calling a virtual method in a superclass, then we regard it has usage dependencies on all the subclasses of the superclass. For example, the class *EvolutionRunner* has usage dependencies on the subclasses of the class *Mutate* such as *ValueMutate* and *BitMutate*.

(4) The three methods generally perform well on component packages, but HC is better.

Component packages in Figs.1 and 3 are cohesive. In Fig.1, the package *PCompiler* is cohesive according to the discussion in Section 2.1. In Fig.3, the package *Population* consists of the classes that represent evolutionary objects in the domain of evolutionary computing. The package *IndividualSet* encapsulates the concept of solutions. In this package, the class *IndividualSet* and *Individual* express solutions for problems, and the class *IndividualDescription* and *ValueRange* descript the characteristics of the solutions. The function of the package

*Builder* is to create the objects of the class *EvolutionRunner*. And the package *Runner* having only one class is obviously cohesive.

According to Fig.6, the RCs of the component packages are much greater than 0, which indicates that RC can be applied to component packages. Due to lack of maximal value, however, the RCs often can not tell engineers whether a package is cohesive or not. But it can be used to compare the cohesive degree of packages. For example, RC(*Runner*) is higher than RC(*IndividualSet*), which illustrates that *Runner* is more cohesive than *IndividualSet*.

Besides in Fig.6, the average of the EECs is 0.61, no longer 0. This shows that EEC is suitable for component packages. However in Tables 1 and 2, we find that EEC(*PComplier*), EEC(*IndividualSet*) and EEC(*Builder*) are 0.46, 0.23, 0.33 respectively, which are low. This is because that the relation number in the packages is much smaller than that in a corresponding complete graph which is the standard in EEC. For example, there are only two relations in the package *IndividualSet*. But the corresponding complete graph has 6 relations.

Also in Fig.6, the HCs are much higher than the RCs and EECs, except the package *Population*. But in terms of Table 2, HC(*Population*) = 0.88, which can shows that *Population* is cohesive. Unlike RC, HC has maximal value. Besides, HC can usually work effectively on the cohesive packages with the low EECs, such as *PComplier*, *IndividualSet* and *Builder*. Therefore, HC usually works better than RC and EEC on component packages.

As shown in Table 3, HC performs effectively on applications by comparing other package cohesion metrics, which indicates that the basic idea of CRC is feasible. Firstly, the metrics without consideration of inter-package dependencies such as RC and EEC are only applicable to component packages in general. The reason is the gap between high-level semantics and low-level dataflow. Secondly, the application areas of HC are extended to interface, implementation and component packages, because HC can reveal semantic relations between classes through client usages. But for utility packages, it requires further study whether HC is applicable to them or not. We believe that the key idea of CRC is rational. Thus, we may need some new CRC measure to assess the cohesion of this kind of packages.

**Table 3** Comparison results of RC, EEC, and HC for applications

| Metric | Package categories | | | | Dependency categories | |
|---|---|---|---|---|---|---|
| | Utility package | Interface package | Implementation package | Component package | Intra-Package dependency | Inter-Package dependency |
| RC | inapplicable | inapplicable | inapplicable | applicable | considered | unconsidered |
| EEC | inapplicable | inapplicable | inapplicable | applicable | considered | unconsidered |
| HC | may inapplicable | applicable | applicable | applicable | considered | considered |

## 6 Conclusions

Packages have a critical role to construct large-scale softwares[1,2]. Cohesion, referring to the relatedness of the elements in a module, can be used to predict the package quality. Traditionally, it is measured by using intra-module dependencies. However, such kind of dependencies is incompetent to the expression of the complex semantics of packages. To solve this problem, we propose a new kind of package cohesion called CRC and define a CRC measure called HC. CRC regards that several classes are related semantically if they are always reused together. The key idea here is to mine the properties of a package in virtue of the behaviors of the package's clients, which is similar to evaluate the usability of a web site via allowing for how users use the site[36,41].

In the future work, we will focus on the following two issues: 1) research the effectiveness of CRC on the packages in libraries by gathering sufficient open source projects; 2) refine the CRC measure to be adapted to more kinds of packages in two ways. First, improve calculation of the measure. Second, refine the kinds of the dependencies among classes by considering the client usages at runtime and the dependency strength. Indeed, usage dependences

can be subdivided by different standards, such as usage frequency at runtime. And different kinds of dependences may well affect the computation of the cohesion of a package.

**References**:


[1] Martin RC. Agile Software Development Principles, Patterns, and Practices. Prentice Hall, 2002.

[2] Lakos J. Large-scale C++ Software Design. Addison Wesley, 1996.

[3] Larman C. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. 2nd ed., Pearson Education, 2004.

[4] Yourdon E, Constantine L. Structured Design. Prentice Hall, 1979.

[5] Chae HS, Kwon YR, Bae DH. Improving cohesion metrics for classes by considering dependent instance variables. IEEE Trans. on Software Engineering, 2004,30(11):826−832.

[6] Briand LC, Morasca S, Basili VR. Defining and validating measures for object-based high-Level design. IEEE Trans. on Software Engineering, 1999,25(5):722−743.

[7] Briand LC, Wüst JK, Ikonomovski SV, Lounis H. Investigating quality factors in object-oriented designs: An industrial case study. In: Proc. of the 21st Int'l Conf. on Software Engineering. 1999. 345−354.

[8] Marcus A, Poshyvanyk D, Ferenc R. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. IEEE Trans. on Software Engineering, 2008,34(2).

[9] Mathias KS, Cross JH, Hendrix TD, Barowski LA. The role of software measures and metrics in studies of program comprehension. In: Proc. of the 37th Annual Southeast Regional Conf. 1999. 13−70.

[10] Zhou YM, Leung H. Empirical analysis of object-oriented design metrics for predicting high and low severity faults. IEEE Trans. on Software Engineering, 2006,32(10):771−789.

[11] Zhou TL, Shi L, Xu BW. Refactoring C++ programs physically. Journal of Software, 2009,20(3) (in English with Chinese abstract). http://www.jos.org.cn/1000-9825/550.htm

[12] Evans E. Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison Wesley/Pearson, 2004.

[13] McConnell S. Code Complete. 2nd ed., Microsoft Press, 2004.

[14] Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1998.

[15] Bieman J, Kang BK. Measuring design-level cohesion. IEEE Trans. on Software Engineering, 1998,24(2):111−124.

[16] Briand LC, Daly JW, Wüst JK. A unified framework for coupling measurement in object-oriented systems. Trans. on Software Engineering, 1999,25(1):91−121.

[17] Bonja C, Kidanmariam E. Metrics for class cohesion and similarity between methods. In: Proc. of the 44th Annual Southeast Regional Conf. 2006. 91−95.

[18] Counsell S, Swift S, Crampton J. The interpretation and utility of three cohesion metrics for object-oriented design. ACM Trans. on Software Engineering and Methodology, 2006,15(2):123−149.

[19] Chae HS, Kwon YR. A Cohesion measure for classes in object-oriented systems. In: Proc. of the 5th Int'l Software Metrics Symp. 1998. 158−166.

[20] Chidamber SR, Kemerer CF. A metrics suite for object oriented design. IEEE Trans. on Software Engineering, 1994,20(6): 476−493.

[21] Hitz M, Montazeri B. Measuring coupling and cohesion in object-oriented systems. In: Proc. of the Int'l Symp. on Applied Corporate Computing. 1995. 25−27.

[22] Marcus A, Poshyvany D. The conceptual cohesion of classes. In: Proc. of the 21st IEEE Int'l Conf. on Software Maintenance. 2005. 133−142.

[23] Chen ZQ, ZhouYM, Xu BW, Zhao JJ, Yang HJ. A novel approach to measuring class cohesion based on dependence analysis. In: Proc. of the Int'l Conf. on Software Maintenance. 2002. 377−383.

[24] Chen ZQ, Xu BW. An approach to measurement of class cohesion based on dependence analysis. Journal of Software, 2003, 14(11):1849−1856 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1849.htm

[25] Zhou YM. Research on software measurement [Ph.D. Thesis]. Nanjing: Southeast University, 2002 (in Chinese with English abstract).

[26] Zhou YM, Xu BW, Zhao JJ, Yang HJ. ICBMC: An improved cohesion measure for classes. In: Proc. of Int'l Conf. on Software Maintenance. 2002. 44−53.

[27] Zhou YM, Wen LJ, Wang JM, Chen YJ, Lu HM, Xu BW. DRC: A dependence relationships based cohesion measure for classes. In: Proc. of the 10th Asia-Pacific Software Engineering Conf. 2003. 215−223.

[28] Zhou YM, Lu JT, Lu HM, Xu BW. A comparative study of graph theory-based class cohesion measures. ACM SIGSOFT Software Engineering Notes, 2004,29(2):1−6.

[29] Xu BW, Zhou YM. Comments on “a cohesion measure for object-oriented classes” by Heung-Seok Chae, Yong-Rae Kwon and Doo-Hwan Bae. Software: Practice and Experience, 2001,31(14):1381−1388.

[30] Xu BW, Zhou YM. More comments on “a cohesion measure for object-oriented classes” by Heung-Seok Chae, Yong-Rae Kwon and Doo-Hwan Bae. Software: Practice and Experience, 2003,33(6):583−588.

[31] Patel S, Chu WC, Baxter R. A measure for composite module cohesion. In: Proc. of the 14th Int’l Conf. on Software Engineering. 1992. 38−48.

[32] BW Xu, ZQ Chen, JJ Zhao. Measuring cohesion of packages in Ada95. In: Proc. of the ACM SIGAda Annual Int’l Conf. 2003. 62−67.

[33] Briand LC, Morasca S, Basili VR. Property-Based software engineering measurement. IEEE Trans. on Software Engineering, 1996, 22(1):68−85.

[34] Allen EB, Khoshgoftaar TM, Chen Y. Measuring coupling and cohesion of software modules: An information theory approach. In: Proc. of the 7th Int’l Software Metrics Symp. 2001. 124−134.

[35] Ponisio ML. Exploiting client usage to manage program modularity [Ph.D. Thesis]. Bern: University of Bern, 2006.

[36] Zhou YM, Leung H, Winoto P. MNav: A Markov model-based web site navigability measure. IEEE Trans. on Software Engineering, 2007,33(12):869−890.

[37] Hamming RW. Error detecting and error correcting codes. Bell System Tech Journal, 1950,29(2):147−160.

[38] Shi L, Xu BW, Xie XY. An empirical study of configuration strategies of evolutionary testing. Int’l Journal of Computer Science & Network Security, 2006,6(1A):44−49.

[39] Shi L. Research on test data automatic generation [Ph.D. Thesis]. Nanjing: Southeast University, 2006 (in Chinese with English abstract).

[40] Xie XY, Xu BW, Shi L. A dynamic optimization strategy for evolutionary testing. In: Proc. of the 20th Asia-Pacific Software Engineering Conf., 2005. 15−17.

[41] Borges J, Levene M. An average linear time algorithm for Web usage mining. Information Technology and Decision Making, 2004, 3(2):307−319.

**附中文参考文献**:

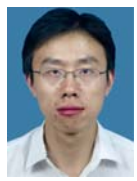
[11] 周天琳,史亮,徐宝文.重构 C++程序物理设计.软件学报,2009,20(3). http://www.jos.org.cn/1000-9825/550.htm

[24] 陈振强,徐宝文.一种基于依赖性分析的类内聚度度量方法.软件学报,2003,14(11):1549−1856. http://www.jos.org.cn/1000-9825/14/1849.htm

[25] 周毓明.软件度量中的若干问题研究[博士学位论文].南京:东南大学,2002.

[39] 史亮.测试数据自动生成技术研究[博士学位论文].南京:东南大学,2006.

**ZHOU Tian-Lin** was born in 1981. She is a Ph.D. candidate at the School of Computer Science and Engineering, Southeast University. Her current research areas are software measurement, program analysis and refactoring.



**SHI Liang** was born in 1979. He is a Ph.D. candidate at the School of Computer Science and Engineering, Southeast University and a CCF senior member. His current research areas are program analysis and testing.

**XU Bao-Wen** was born in 1961. He is a professor and doctoral supervisor at the School of Computer Science and Engineering, Southeast University and a CCF senior member. His research areas are programming languages, software engineering, concurrent software and web software.

**ZHOU Yu-Ming** was born in 1974. He is a professor at the School of Computer Science and Engineering, Southeast University. His research areas are software measurement, program understanding and software maintenance.