Research on Measurement of Software Package Dependency based on Component

Guang-yi TANG
School of Software, Harbin University of Science and Technology, Haerbin, China
Email:tanggy818@hrbust.edu.cn

Hong-wei XUAN
School of Software, Harbin University of Science and Technology, Haerbin, China
Email:henryxuan2005@hotmail.com

Abstract—**Dependence between software packages is of importance to influence the extendibility and stability of system. In existing programs, dependence mainly manifests for class and component. Here, it has the important guiding sense to our system construction and programming. This paper analyzed the dependency between software packages, and designed the algorithm for detection existence of the package dependency loop, based on defined dependency, stability, no-responsibility and stability of the System; and then elevated the package design principles. To validate our design methodology in software development, which is valid and can be helpful for the programmers, we developed a software to analyze the dependencies between the software packages and use a graphical method to express this dependency.**

Index Terms—**Component, dependency, stability, dependency loop, no-responsibility**

I. Introduction

Demand for software quality continues to increase as the software market matures and as competition increases among software producers and vendors. The quality attributes of a software design include reusability, flexibility, understandability, extendibility, maintainability, etc. All these quality attributes are related to software component dependency. Component-based development is the production of software products through the systematic integration of existing software components. The potential benefits of component-based development include increased productivity and quality and decreased cost and time-to-market. The dependency of a software component is a reflection of its external quality, such as reusability and maintainability[1,2].

In general, development of an application system starts with analysis of requirements, following by design, coding, testing, and then delivery[4]. Processes, such as analysis of requirements, system design, program coding, program testing would be repeated, if all start from beginning. It will result repeated work, which may take huge amount of time. Software reuse essentially uses existing software products or engineering knowledge to construct a new system, which avoid repetition in software development.

Various IMS is applied by enterprises according to their applications. Firstly, to extract common points and design as components base on sufficient analysis. Secondly[3], a frame extension point is extracted and encapsulated as independent assembly, including workflow management, alarm management, system management, security management, and system settings. These are common parts of various IMS, which can be inserted as components of a module. Finally, for a specific enterprise that develops software product, they only need to know the logic of business, then load the needs of assembly, according to the system configuration information provided by dynamic, and then call the provided methods and attributes to achieve their specific services functions.

Software component model is the composition of components, which can represent the development of reusable software components and communication between components. By reusing the existing software components, developers can work on fast construction application by using a component object model software as simple as playing building blocks. Therefore, it is not only able to save time and costs, improve work efficiency, but also produces more standardized and reliable application software[7,8].

II. The INFLUENCE of the package dependencies

A. Software Coupling

Promoting reuse is an important factor in object-oriented software design. There are several reuse approaches for object-oriented software, including: component, design patterns and frameworks. Design patterns are template solutions for solving a range of recurring problems. Frameworks are reusable partial applications that can be specialized to produce custom applications. Both design patterns and frameworks provide skeleton solution to a problem. Developers are

© 2012 ACADEMY PUBLISHER
doi:10.4304/jsw.7.9.2091-2098

responsible for plugging in the specific code, according to the needs for solving problems. This paper focuses on component reuse rather than design patterns or frameworks.

Coupling is a measure of the degree of interactions between two software components, such as classes, modules, packages, etc. A good software system should have high cohesion within each component and low coupling between components[5,6]. Strong coupling, indicating a high degree of dependency between software components, may affect the quality of a system[9].

Coupling between two components strengthens the dependency of one component on the other[10], and increases the probability that changes in one component may adversely affect other components. This makes software maintenance difficult and the likelihood of regression faults greatly increase. Strong coupling induces strong dependencies between software components, thereby hampering software reuse. For example, if a software component has many dependencies with other software components, it may be impossible to reuse this component in a new product without either (1) incorporating it together with other related components, or (2) redesigning and reimplementation this component to remove such dependencies. While option (1) may result in redundant reuse, option (2) may cause modifications to the functionality of the component. Hence, either of these two approaches are contrary to the spirit of component reuse.

B. *Software Package Dependency*

Along with the increase in size and complexity, higher level of organization is required[11]. Class is the organizational unit that can be used conveniently for small applications. For large applications, particle size will be too small, if only use class. Therefore, we need a "bigger unit" to support a large application organization, which is called “package”. Dependencies between classes are often across boundaries of packages[12]. Hence, dependencies also occur between packages. These relationships have to be managed due to the dependencies between packages that represent a high application level organizational structure.

Package design is the most important step for publishing a package, especially for large-scale application that contains a lot of combinations. If classes are categorized into a package only due to their similarity, it is possible to produce a bad package structure and difficult to be published[13,14]. It also results difficulties in reuse, changes and other issues. Therefore, principles are needed for classification. These principles will be used for dealing with relationships between packages and measuring the degree of participation.

Occurrence of the ‘morning after syndrome’ has to be effectively avoided, in order to improve the stability of packages and reduce the dependency, when publish a package. The occurrence of ‘morning after syndrome’ is due to changes in dependencies, which results in a unstable foundation. Dependencies must make appropriate changes. Using the package distribution mechanism, dependent must choose a specific version of a published package. When the package is published, its contents are not allowed to change, hence, dependencies are also unchangeable. Simultaneously, any changes of dependent package will be published as a new version. And dependent can choose or abandon the new version.

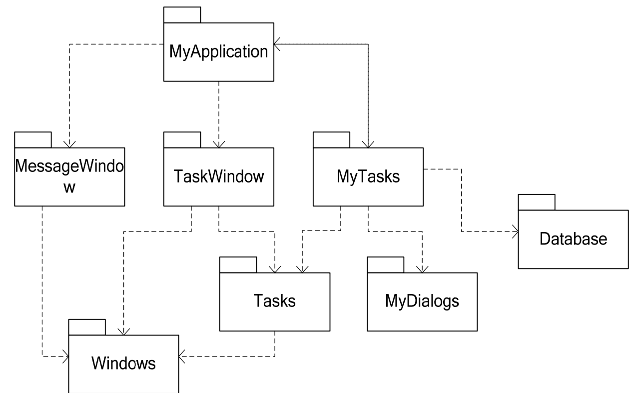

Figure 1. Software package dependency

Figure1 illustrates the structure for a typical application package based on component. It shows that structure of package dependency is more important than functions of applications. The structure is a directed graph. Package is a node, while dependency is a directed edge.

In Figure 1, if you need a new force to change package MyDialogs to use package MyApplication. This creates a dependency loop, the loop dependency will lead to some direct adverse consequences, for example, the developer who works in MyTasks package in order to release MyTasks package, they must have compatible Tasks, MyDialogs, Database and the Windows package. However, the dependence of the ring exist, they must now also compatible MyApplication, TaskWindow and MessageWindow package, that is, the package MyTasks is dependent on all other packages in the system now. The result of this dependency loop is that the package MyTasks is difficult to release, package MyDialogs have the same problem. In fact, the dependency loop will force the MyApplication, MyTasks and MyDialogs packages are release always at the same time. They have actually become one big. Thus, all developers who work on these packages will once again suffer the ‘morning after syndrome’. Their mutual release action must to be completely consistent, because they must use each other the same version exactly.

III. SOFTWARE PACKAGE DEPENDENCY METRIC METHODS

A. *Dependency*

Definition 1 dependence: if in the software package P1 (C1, C2, ... Cn) and P2 (C1, C2, ... Cn)

$$\exists(p_1(c_i) \to p_2(c_j)) or \exists(p_2(c_i) \to p_1(c_j)),$$

called software package P1 and P2 having dependence, if

© 2012 ACADEMY PUBLISHER

$\exists(p_1(c_i) \to p_2(c_j))$, then P1 relies on the P2, if $\exists(p_2(c_i) \to p_1(c_j))$, then P2 relies on the P1, if $\exists(p_1(c_i) \to p_2(c_j)) and \exists(p_2(c_i) \to p_1(c_j))$, called P1 cycle dependent P2, or P2 cycle dependent P1. Wherein, said P1 P2 software package, C1, C2, ... Cn means the classes which in the software package, the arrow indicates the software package in the kind of coupling relationship, including, inheritance of class, class reference, class extension, not including the implementation of the interface.

In this article, we discuss the dependency between the packages, so here is the dependence between different classes of dependent packages. The dependencies between classes in the same package are not in the scope of this article.

B. Stability

We can calculate the number of the dependencies which into and out of the packages to measure the stability of a package.

a. Afferent Coupling(Ca): The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.

b. Efferent Coupling(Ce): The number of other packages that the classes in the package depend upon is an indicator of the package's independence.

c. Instability(I)

$$I = \frac{C_e}{C_a + C_e} \quad (1)$$

This metric is an indicator of the package's resilience to change. The range for this metric is 0 to 1, with I =0 indicating a completely stable package and I =1 indicating a completely instable package.

Definition 2 A package is stability if and only if I =0.

There is an example in Figure 2:

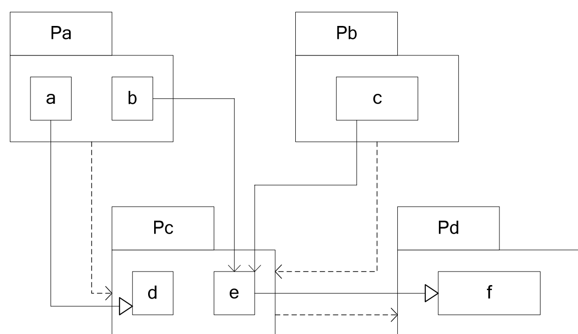

Figure 2. Diagrams of Ca, Ce, and I

The value of Ca is number of dependencies which depends on the classes in the package, and we can calculate the measure of Ce that the number of dependencies which in-house and depends on the classes outside the package.

The dashed arrow expresses the package dependency, the solid arrow indicates the dependency between the classes, and these express how to constitute the dependency between the packages from the solid arrow.

Now, we use the formula (1) to calculate the stability of the package Pc. There are 3 outside dependencies which depend on the classes in the package, so Ca=3.And there is a dependency which is depended by the class in package Pc, so Ce=1, I =1/4.

In C + +, these dependencies are generally represented by include statements. In fact, if we put one class source code into a package, then the calculation of measuring I is quite easy, but it will cause many packages. In Java, you can calculate the import statement and class modification to calculate the number of names to measure the value of I .

When the value of I is 1 for a package, it means there is no other package relies on the package (Ca = 0), while the package is dependent on other packages (Ce> 0), then this is a package of the most unstable state: it is no responsibility but dependent on others. Therefore, the package will often change because there is no package depends on it and the dependence of package will provide a wealth reasons for change.

On the other hand, it means that the other package will be dependent on the package when the value of I is zero (Ca>0), but the package dose not depend on any other packages (Ce=0). So the package is the responsibility and no dependence, and this package has reached the maximum degree of stability. It depends who makes it difficult to change, and no dependencies will force it to change.

Stable-dependencies principle (SDP) provides the metric value of I should be greater than its dependencies the metric value of I (in other words, the value of I should follow dependent decreases in the direction).

If the classes in the package are some of the basic class, they did not inherit other classes; or they are inheritance the classes which the system development environment is provided, we called stable-class. The classes which is provided by the develop system will not change in general, so we also called these classes stable-class. For example: in the VC++ development, if our class is inherited from the MFC, and in the java development environment, our class inheritance the class from package java.lang, we are called stable-class.

C. No-responsibility

Design can not be completely fixed because make the design to maintain some level of volatility is necessary, and this can be achieved by following the Common Closure Principle (CCP). We can create a certain type of change-sensitive packages that are designed instable package by using this principle. For any package, it should not be depended by the packages which are difficult to change, otherwise, the package is also difficult to change.

© 2012 ACADEMY PUBLISHER

Definition 3 A package is no-responsibility if and only if it is not used by any other packages.

If the classes in a package are some of the final class, and the classes are not inherited by class in the other packages, these classes are not claimed responsibility. In system development, these classes can change at any time.

D. Depend Loop

Definition 4 dependent loop in the system software package developed by P1, P2, P3 ... ... Pn, if $\exists(p_i \rightarrow p_j, p_j \rightarrow \ldots\ldots p_i)$, called the package P1, P2, P3 ... ... Pn in a dependency loop.

If the dependency loop exists in the system development packages, then the system stability and scalability will be facing a serious problem, and it will inevitably produce the 'morning after syndrome'. Among these packages, as long as the contents of a package of changes have taken place, then along with other packages should be modified, which takes a heavy workload, and the changes in the development of software which is inevitable.

We can use linear algebra in the matrix calculation to determine the system software package is dependent on the presence of loop[12].

Set the package dependency graph G is the adjacency matrix A, then G does not contain dependency ring if and only if A，A2，A3，……，An of the n diagonal matrix elements are zero. For example, the dependence of Figure 1, we can use the following adjacency matrix representation.

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Here, a_{ij} represents that the dependency between package a_i and package a_j. And the matrix A is a symmetry matrix.

Prove the following:

Let the dependency graph G is a directed graph, A is the adjacency matrix of G, a (i, j) is A's row i column j element which represents the number of chain vi to vj of length 1.

a2(i, j) is the element of A2 row i column j: a2(i, j) = a(1, i) * a(i, 1) + a(2, i) * a(i, 2) + … + a(n, i) * a(i, n)；

Clear, vi to vj of length 2 each chain as two chains of length 1 is connected, so there are:

a(1, i) * a(i, 1) + a(2, i) * a(i, 2) + … + a(n, i) * a(i, n)，that is a2(i, j)；

Same reason, ak(i, j)represents that vi to vj of length k, the number of chain.

In particular that i = j;

So, ak(i, i)through vi of length k, the number of rings;

Obviously, if the n vertex of G there is a ring, there must be a length of less than n ring, therefore, the G does not contain the ring if and only if A，A2，A3，…，An, the diagonal of these n matrix elements are 0.

It can also be obtained by proving that if the diagonal elements of the matrix An are not all 0, then the matrix A must exist in the length n of the loop. Moreover, these loops contain diagonal elements are not zero.

For example, Figure 1 adjacency matrix A, is illustrated in Figure 1, there is a dependency loop.

$$A^2 = \begin{bmatrix} 3 & \ldots\ldots \\ \ldots\ldots & \ldots\ldots \end{bmatrix}$$

In view of this, the algorithm can be designed to detect the following loop

```
boolean ifexistcircle(A[n][n])
{
  Int M[n][n],C1[n],C2[n];
  int i=0;
  while(i<n)
   {
    M=M*A;
  if(the diagonal elements of M are not all 0)
  {
  Save the diagonal nonzero elements of M to C1;
  break;
}
      i++;
}
```

Traverse the A belongs to the C1 elements, and all the length of i loop save to C2;

```
if(C2 is empty)
   return true;
else
   return false;
}
```

On the collection returned to judge, if an empty note there is no loop; if it is not null notes there are loop in the graph, we can walk through the collection to loop processing.

IV. Dependent Loop Elimination

We can lift the dependencies between packages and get the dependency graph back to directed acyclic graph under any circumstances, and this is mainly in two ways:

One way is to use the Dependency-Inversion Principle (DIP), another way is to create a new dependency, and these two methods will be described in detail with diagram.

The first method is to use the Dependency-Inversion Principle (DIP). In view of the situation in Figure 4, you can create an abstract base class which MyDialogs's interface need, then put the abstract base class into MyDialogs and make MyApplication class inherit this abstract base class, which reversed the dependencies between MyDialogs and MyApplication, so that a

© 2012 ACADEMY PUBLISHER

dependency loop is lifted in the graph, an example shown in Figure 3.

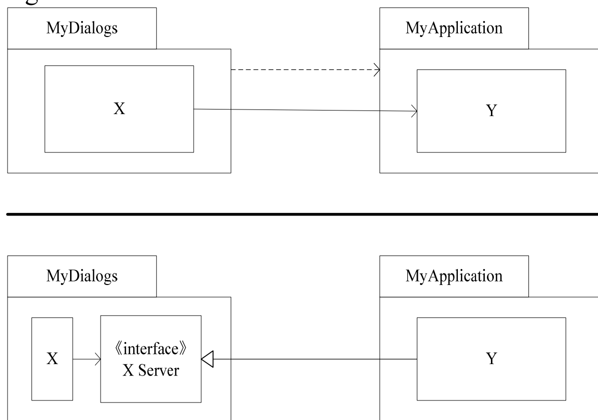

Figure 3. Using DIP lift dependent loop

Please note that from a customer perspective rather than the service's point of view to the named interface, it is because the interface is part of the customer rather than services, and this is a good programming practice.

The second method is to create a new package which MyDialogs and MyApplication are dependencies, and then put the classes which MyDialogs and MyApplication are dependencies into this new package, as shown in Figure 4:

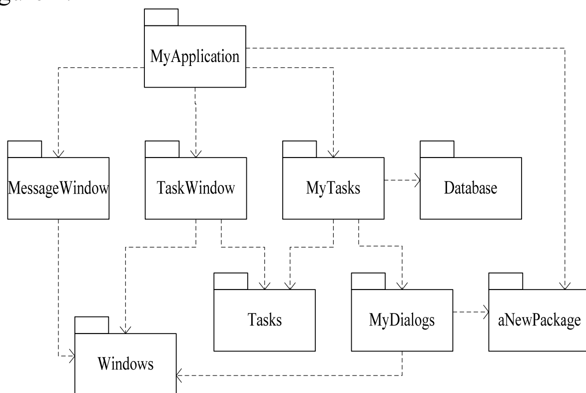

Figure 4. Using new package lifting dependent loop

If the packageP1,P2,⋯⋯Pn, there are dependency loop, it can be expressed as Figure 5.

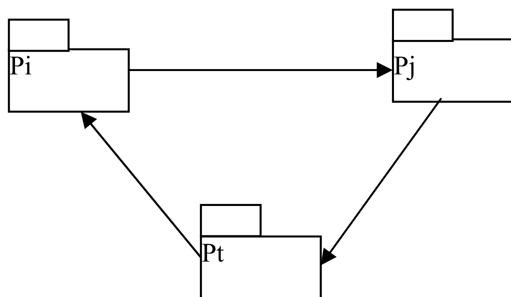

Figure 5. Software package dependency loop

Assume:

$$p_i(c_i) \to p_j(c_j), p_j(c_j) \to p_t(c_t), p_t(c_t) \to p_i(c_i)$$

then we can add a new interface class C_Δ in package p_i, let $p_i(c_i) \to p_i(C_\Delta)$, $p_j(c_j)$ implementation the interface $p_i(C_\Delta)$ .then these dependency relationship can be expressed as in Figure 6:

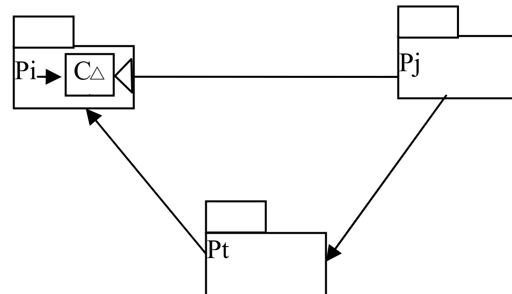

Figure 6. Dependent loop elimination

At this point, the existence of which can eliminate the dependence loop, this method is known as Dependency Inversion, using this method can eliminate the dependence of the entire ring. Another simple method is to add an extra package to their common dependence on this new class package, so they rely on this new package.

Definition 4 System is stable if and only if the system does not exist depend loop in the development of software packages.

There is not absolute stability system in our development of the system, we say stability is relative. It is stable for a thing if "it is not easy to be moved" in Webster Opinion, stability relate with the work of the change. Coin is not stable, because the effort required to tear down it is very small. However, the table is very stable; because it is down to spend considerable effort. Our software package can be dependent on the package which the system development environment provided, and system development environment provided by the software package is stable, that is, in a fairly long period of time, will not change, our own developed software the basis on which package is fixed, then our system is stable.

Definition 5 System is unstable, the system's package dependency loop exists, and then we say the system is unstable.

If there is dependency loop in the development of the system, then relies on a ring in any class package changes will affect other packages dependent on ring and a chain reaction occurs, the system maintenance and system expansion will bring very serious consequences.

We do not hope that all of packages are the largest degree of stability, because if that the system is not able to change. In fact, we want to get out of the structure of packages in design is that some of the packages is stable and the others not.

The packages often changed are put at the top and depend on the packages in the bottom. It is a useful agreement for put the unstable package at the top, because the up arrow means that any violation of the SDP, that is the value of I underlying package cannot exceed the upper package value.

© 2012 ACADEMY PUBLISHER

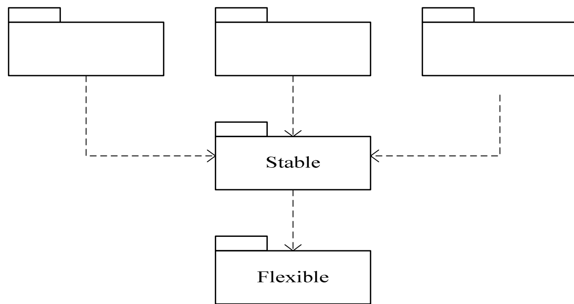

Figure 7. Violation of the SDP

It shows an example would violate the SDP in Figure 7, the values of Ca, Ce and I are show in table I, we can that the package Flexible is more stability than stable, but in the application the developers often modify the package Flexible, so now we hope Flexible is unstable because we want to change it easy. However, it is violation of the SDP that the developers who work on the package Stable create some dependencies on the package Flexible, because the value I of Stable is smaller than the value of Flexible in the table Ⅰ. Then Flexible is not easy to change, and changes to the Flexible force to deal with

TABLE I.
THE CA CE AND I OF STABLE AND FLEXIBLE

package	Ca	Ce	I
Satble	3	1	1/4
Flexible	1	0	0

the change of the Stable and all its dependent effects.

It must be some way to lift Stable dependence on Flexible to fix this problem. Now suppose that the class C in package Flexible is depended by the class U in package Stable, and we can use Dependency-Inversion Principle (DIP) to fix the problem. Firstly, create a interface class IU and put it in the package UInterface, and to ensure that all the functions are declared in the interface class IU which the class U will use. And then, let class C inherit from IU, as show in figure 8, and this lifts the Stable dependent on Flexible and prompted the two packages are dependent on UInterface. The package UInterface is very stable (I = 0), but Flexible still keeps it required instability (I = 1), as show in table II. Now all depend on the direction is reduced along the I

TABLE II.
THE CA, CE AND I OF STABLE AND FLEXIBLE

package	Ca	Ce	I
Satble	3	1	1/4
Flexible	0	1	1
UInterface	2	2	0

direction.

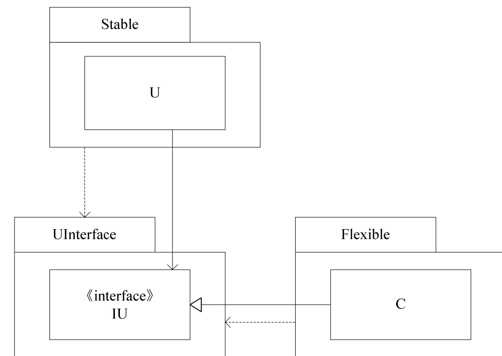

Figure 8. Using DIP to lift non-compliance stability

A dependency should be refractor if its contribution to system functionality (importance) is small compared to its negative impact on testability. The best time to evaluate the importance of a dependency and to refractor it is during the design stage, followed by the time when the dependency is introduced into the system. Therefore it is important to provide immediate feedback to designers and developers about the effect of a new dependency. During maintenance the effort to refractor test-critical dependencies is likely larger but necessary and worthwhile.

We must consider the contradictions between the reusability and the scalability when choice the class put into the packages. That is not a simple work for the balance between these two, and this balance is always dynamic. In other words, the division now appears appropriate to the next year may be no longer appropriate. Therefore, when the project focus from scalability to reusability, then the package structure will be subject to change.

V. PACKAGE DESIGN PRINCIPLES

We all have known that packaging design can make or break a product. This is especially true in the case of new products being released onto the market. Effective and functional package design can be the single point alone that can inspire a consumer to choose your product in favor of another. This is where a packaging design strategy comes into your marketing mix. We recommend that you seek expert advice when designing your packaging to ensure your product is having the most impact possible on consumers. The following examples should be included as a standard as part of any packaging design strategy.

(1) Acyclic Dependencies Principle, in the package design, not dependent on the presence of ring, otherwise the system is difficult to maintain.

It will happen the 'morning after syndrome', if there are multiple developers are also changing the same set of source files in the development environment. That is not a serious problem in a small program with only a few developers, but in a large-scale project with many developers, the 'morning after syndrome' will bring terrible nightmare. It cannot build a stable version of the project in a few weeks in the team which lack of discipline is very common. Instead, everyone is too busy

© 2012 ACADEMY PUBLISHER

to change his or her code repeatedly, trying to make it compatible with other's recent changed.

(2) No responsibility for the package design principles, for some package which often change, we can design it for no liability package.

For system development, they are often changed the application business classes (mainly used to handle the user's business requirements) which are generally putted in a package has no-responsibility, because they will be changed with the user's business environment change in special application.

(3) The principle of stability of the package, the bottom of the package must be stable in the package dependencies.

We write programs are generally not completely from scratch in system development, but on certain foundation for system development. For example: development environment provides us with some basic class package java.util, jaba.swing etc.. There are also many classes are not likely to change frequently in our writing program, for example: some kinds of database access classes, the classes of workflow definition and so on, they are stable for a long time in some applications.

(4)The stability of the system principle, we must ensure that the system is stable in our design of the software packages.

We must ensure the system developed is stability no matter what method is used, that is, the package at the upper must be non-responsibility and the user can change it in any time according to the user's business requirement change, and the package in the bottom must be stable which does not allow users freely modify. The packages in the middle called the restricted package, we should always pay attention to these package changes and assess their possible consequences of change.

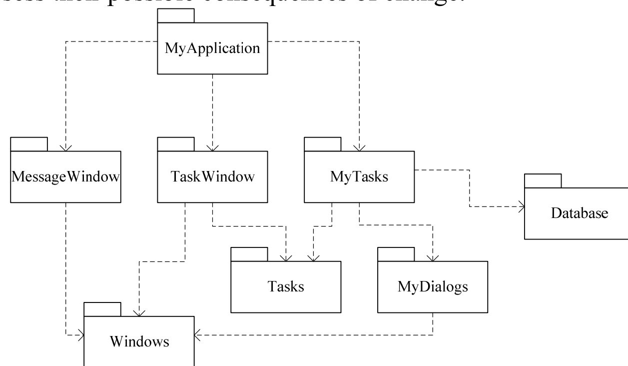

Figure 9. The ideal software package dependency graph

As shown in Figure 9, is our software package of a typical structure design. The figure shows the composition of a very typical application package structure. The bottom two pack Windows and Database which are provided by the system development environment, and they are stable; and the package MyApplication which is the most easy to change in the system, and it is a no responsibility package.

VI Application

A software is developed to evaluate our designed method, which also bring the help for programmers. The software analyzes between software package's dependence, and expresses using the graph method. This software has functions in several aspects:

1. Import project, the option includes a project path, which is used to introduce source file that will be measured.

2. Edit project, the option includes engineering analysis and generate dependency graph. The function of engineering analysis is to analyze the introduced source file, then converted to the defined data structure, and then generate dependency graph.

3. View, functions includes zoom in and zoom out, which is used to adjust the size of dependency graph, in order to facilitate the viewing of the whole or partial information.

4. Help is the function to provide a brief introduction to the system.

During the implementation, coupling method is used to calculate the package input and output. In the method, a list is added into each class, and introduces other class by record the classes (if introduce a package with multiple classes, there will be many records of the package). It converts the dependence between the classes to the ones between class and package.

To construct a dependence graph, we iterate through list of each class. The introduced record will make weight of the package plus one. Hence, it will convert dependency between class and package to dependency between the packages. We can calculate the in-degree and out-degree in the directed graph, when the statistics of a package afferent coupling and efferent coupling.

For instance, project1 is an assumption, it consists of six packages: main.UI, main.framwork.java, main.framwork.factor, main, main.framwork, and main.framwork.basic. Classes in the package of main.framwork.basic are mainly basic ones, such as establishment of the connection, data packet processing, and data access etc. The package of main.framwork is a main application of the framework for the whole system. The package of framwork.factor mainly deals with operations for user services, for user interface and data connection. The package of main.UI is mainly deal with operations for interface, that for communications between users. The package of main.framwork.java mainly include the classes of business entity, completing the main work of the system. The package of main is the upper application of the system, along with user's needs, which is the fastest package changing in the system.

In summary, the package of main has to be a no-responsibility package, while the package of main.framwork.basic should be a stable package, and the other four should be restricted package. It is important to pay attention to their impact of change, the dependency graph is shown in Figure 10.

© 2012 ACADEMY PUBLISHER

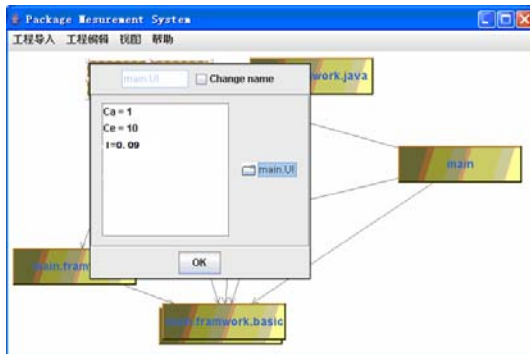

Figure 10. The dependency graph of project1

In this figure, when we clink on an icon of a package, it will show the afferent coupling(Ca) , the efferent coupling(Ce) and instability(I).

VII. Conclusions

It is very important for our system success of the software package design in the development based on component. We must analyze the dependence of the package in the first, and divide the packages into stable packages and no-responsibility packages, and the others into the restricted packages, in the development of which we should pay close attention to changes in these packages to construct a stable system.

Quality is one of the most relevant properties of the software. Dependency is one factor that can affect software package quality. To assess a software package quality, we need to consider many other factors, such as the design principle provided by Martin. Following the guidelines described in this research is an important step toward designing high quality software package. We believe that the study presented herein should encourage other researchers and tool developers to adapt and use this classification in combination with other design principles to develop more accurate software package dependency metrics that can help to effectively predict and measure several other software quality factors.

Acknowledgment

This work was supported by the Scientific Research Foundation for Returned Overseas Chinese Scholars, State Education Ministry, Heilongjiang Province Natural Science Foundation (LC2009C19).

References

[1] HuangWangen, Chen Songqiao. The software architecture and its complexity metrics based on component operations. Computer Engineering and Applications, 2007,43(14): 66-70.

[2] LiJinhua, Guo Zhenbo, Zhao Yun, Towards quantitative evaluation of UML based software architecture. Proceedings of the 8th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing. Washington, DC: IEEE Computer Society, 2007. 663 - 669.

[3] Liguo Yu. Understanding component co-evolution with a study on Linux[J] Empirical Software Engineering, 2007,12, (2) .

[4] QIAN Guan-qun ZHANG Li ZHANG Lin Philip Lew. Modeling Method and Characteristics Analysis of Software Dependency Networks[J]. Computer Science. 2008.11(35):239~243.

[5] Schaeh S R,Jin B,Wright D R et al. Quality impacts of clan-destine common coupling .Software Quality Journal, 2003,11, 11 (3) :211～218 .

[6] Paul Dourish;Rebecca E. Grinter;Jessica Delgado de la Flor;Melissa Joseph; Security in the wild: user strategies for managing security as an everyday, practical problem [J]. Personal and Ubiquitous Computing, 2004.

[7] Chang,Z,X Mao,Z.Qi. Component Model and Its Implementation of Internetware Based on Agent .Ruan Jian Xue Bao(Journal of Software), 2008,19, 19 (5) :pp.1113-1124.

[8] Allison Sall;Rebecca E. Grinter; Let’s Get Physical! In, Out and Around the Gaming Circle of Physical Gaming at Home [J]. Computer Supported Cooperative Work (CSCW), 2007.

[9] Ruzhi XU; Dongsheng CHU; Zhikun ZHAO; Kang-Kang ZHANG. An Architecture for Agent-based Distributed Component Repository . International Seminar on Business and Information Management, 2008: 116~120.

[10] Xiaocong Wang;Jing Liu;Xigen Huang;Lihong Men;Minjie Guo;Donglan Sun; Controlled Synthesis of Linear Polyaniline Tubes and Dendritic Polyaniline Fibers with Stearic Acid [J]. Polymer Bulletin, 2007.

[11] Fan Jiang;Shi Liu;Jing Liu;Xueyao Wang; Measurement of ice movement in water using electrical capacitance tomography [J]. Journal of Thermal Science, 2009.

[12] Kupke C,RuttenJ. Observational coalgebras and complete sets of co-operations[J]. Electronic Notesin Theoretical Computer Science, 2008,203, 203 (5) :153-174 .

[13] Weifeng Pan;Bing Li;Yutao Ma;Jing Liu;Yeyi Qin; Class structure refactoring of object-oriented softwares using community detection in dependency networks [J]. Frontiers of Computer Science in China, 2009.

[14] Yutao M, Keqing H, Du Dehui. A Complexity Metrics Set for Large-scale Object-Oriented Software Systems[C] .Com-puter and Information Technology,2006.CIT’06. The Sixth IEEEInternational Conference on. 2006:189.

Guang-Yi Tang is a lecture in School of Software, Harbin University of Science and Technology, China. He was born on August 1980. He achieved his Master degree in computer software and theory in 2007 at the central china normal university. His research emphasizes on workflow,interoperation and open-source software development.

Hong-wei Xuan is a lecture in School of Software, Harbin University of Science and Technology, China. He achieved his Master degree in Department of Computing, University of Bradford in 2005. His research emphasizes on software engineering and natural language processing.

© 2012 ACADEMY PUBLISHER