ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.3, March 2009, pp.597−607
doi: 10.3724/SP.J.1001.2009.00550

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

重构 C++程序物理设计*

周天琳[1+], 史 亮[4], 徐宝文[1,2,3], 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)

Refactoring C++ Programs Physically

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

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

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

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

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

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

Zhou TL, Shi L, Xu BW, Zhou YM. Refactoring C++ programs physically. *Journal of Software*, 2009,20(3): 597−607. http://www.jos.org.cn/1000-9825/550.htm

Abstract: This paper proposes physical refactoring and digs into its process and methods. Physical refactoring is a disciplined technique for restructuring the physical structure of a software system, to improve the efficiency of software development, while preserving the system's external behavior. It follows the best practices of refactoring to change the system in small and iterative steps, and applies refactorings according to the standards of physical design. Case studies demonstrate that physical refactoring may continuously improve software quality from the viewpoint of the physical structure.

Key words: C++; software evolution; software refactoring; physical design; software reengineering

摘 要: 整合重构的基本思想和物理设计的基本技术,提出了物理重构的概念.它是对软件物理结构的再设计,目的是在不改变软件外在行为的前提下,调整软件组织结构,从而提高软件的开发效率和可维护性等.在此基础上,提出用"识别-重构-评估"的迭代过程来实施物理重构,并介绍了常用的物理重构方法.实例研究表明,物理重构能够有效地优化系统的物理结构,使开发者从多个角度持续改善软件质量.

关键词: C++;软件演化;软件重构;物理设计;软件再工程

中图法分类号: TP311 **文献标识码**: A

* 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 National Basic Research Program of China under Grant No.2002CB312000 (国家重点基础研究发展计划(973)); the National High-Tech Research and Development Plan of China (国家高技术研究发展计划(863))

Received 2006-08-07; Accepted 2007-09-30

1 Introduction

As for large-scale software, software architecture is one of the key factors affecting the efficiency of software development[1], which has both a logical and physical structure. Good architecture keeps the software modules high-cohesion and low-coupling, which helps developers work independently and consistently. Nevertheless as the software is modified and enhanced to be adapted to new facts, better understanding, changing requirements, and so on, the architecture becomes more complex and drifts away from its original design, thereby dissatisfying the requirements of the software development and maintenance[2−4].

Many methods are proposed to preserve the architecture in its evolution, of which refactoring is a widespread and effective technique. However, the existing refactoring techniques mainly concern themselves with the logical structure[2−5], paying little attention to the physical structure. The logical structure involves logical entities and logical relationships, such as classes, inheritance, etc., while the physical structure consists of components, packages, compile-time dependencies, etc. Good physical structure can improve the efficiency of team development, and reduce the cost of software build, integration, testing and release, which counts with the construction and maintenance of large-scale software[6,7]. The research domain that focuses on the design of the physical structure is referred as *physical design*, whose goals include partitioning software system into reasonable packages, reducing build cost, enhancing testability and team development efficiency, etc. But physical design also fails to support the continuous optimization of the physical structure in software evolution.

For the sake of preserving a good physical structure, this paper integrates the basic ideas of refactoring and the main techniques of physical design to propose the concept of physical refactoring, and then discusses its basic process and methods. The contributions of this paper include:

1) We put forward a new kind of refactoring that can help developers optimize the physical structure in the evolution of large-scale software. Compared to the existing refactoring techniques, physical refactoring converges on the physical structure, which is of great importance to the development, maintenance and reuse of large-scale systems. Compared to physical design, it is able to continue to improve the physical structure, which helps to fit the structure for the changing development environment.

2) We review the existing techniques of physical design to propose a catalog of physical refactorings. In the catalog, we not only describe the name and action of the refactorings, but also link bad smells to refactorings. Herein, *bad smells* are undesirable physical structure that violates the standards of physical design. This catalog provides developers with guidelines on when and where to refactor physically and how best to refactor physically. Moreover, it enhances the level of communication among developers.

As a popular language for system development, C++ is widely applied to construct large-scale software. Consequently, this paper is restricted to the large-scale C++ software development. But the techniques discussed below are still valuable for the software systems written in other programming languages.

The remainder of this paper is structured as follows: Section 2 analyzes the pervasive problems in the development of large-scale C++ programs by means of an illustrative example. Section 3 proposes the concept of physical refactoring and provides an overview of its process. Section 4 summarizes and presents some common physical refactorings. Section 5 discusses related work. Section 6 concludes.

2 Problems in Large-Scale C++ Software Development

For large-scale software, the physical structure plays a vital role in keeping the independency and consistency of development work, which is crucial to team development efficiency. In this section, we introduce an LAN

simulator (named as NetSimulator) that will be used throughout this paper to discuss the importance of the physical structure and show the process of physical refactoring. NetSimulator simulates star and ring network and its implementation uses Bridge pattern[8]. Although NetSimulator is a simple example, it is enough for us to illustrate why and how to refactor the physical structure. Furthermore, its simplicity makes us focus on the key problems in the development, and safely ignore some unnecessary complexities of large-scale systems. Figure 1 shows NetSimulator's initial physical structure that is composed of components, packages and compile-time dependencies:

• **Component**. A *component* is a basic unit of physical design, usually recording a class's declaration and implementation. In C++, a component often corresponds to a pair of a header file and a dot-C file. Especially, if a component records an abstract class, it is called an *abstract component*, or else it is a *concrete component*. In Fig.1, Net is an abstract component, while Star is a concrete one. In practice, such correspondence between class and component incarnates a kind of best practices.

• **Package**. A *package* is a set of semantic-correlated components or packages, which is taken as the basic unit for the assignment of development work. In C++, a package corresponds to a file folder. Similar to component, a package consisting of abstract components is defined as an *abstract package*. In this paper, both components and packages are called *physical entities*. Figure 1 contains two packages PNet and PElem.

• **Compile-Time dependency**. There are *compile-time dependencies* among physical entities: if the compilation of a physical entity E_1 requires the header files of another physical entity E_2, then there is a compile-time dependency from E_1 to E_2. In C++, the compile-time dependencies are caused by using #include directives. For example, the component Star uses #include"Net.h"in its header file Star.h to import the component Net's header file Net.h, so there is a compile-time dependency from Star to Net.

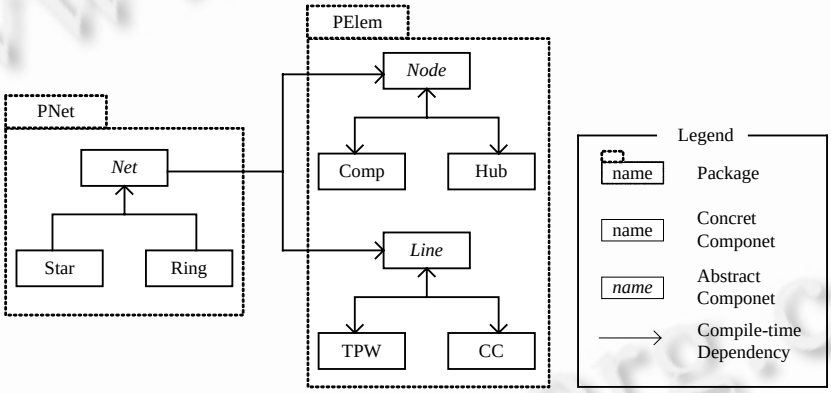


Fig.1 Initial physical structure of NetSimulator

The initial physical structure of NetSimulator is not capable of fitting the changing development situation. Suppose at the initial stage of development, the NetSimulator team has two developers: Mary and Tom. Mary develops the package PNet, and Tom answers for the package PElem. As the increase in the system scale and complexity, Jim joins in the team to speed up the development, bringing some problems:

First, the initial structure makes it hard for developers to work independently. Since package is the unit of development work, the system composed of two packages is unaccustomed to the three-person team. Additionally, the concrete components are the implementation of the package PElem, thus their modification should not affect any physical entity except PElem. But it is not the case in Fig.1. This causes that Tom and Jim may give up some reasonable modifications in consideration of Mary. It is hence important to reorganize unreasonable package structure.

Second, the initial physical structure makes it difficult for developers to coordinate their works. Software build

is an important means of guaranteeing the consistency of the team[7]. Nevertheless in Fig.1, the class Node (Line) is responsible for creating objects of its subclasses, so that it depends on the whole definition of its subclasses. This results in cyclic dependencies that aggravate the incremental build cost when modifying the package PElem. Therefore, removing undesirable dependencies is one of the main tasks for the large-scale software development.

As mentioned above, to improve team development efficiency, the physical structure should being maintained to suit the change of the development situation. However, the existing refactoring techniques focus on logical design, which have difficulty to solve the above problems that NetSimulator faces. In addition, although physical design can ameliorate the physical structure, most researches in this field do not refer to the continuous betterment of the physical structure in software evolution. So a new technique for resolving these problems is urgently called.

3 Physical Refactoring

3.1 Basic ideas

To solve the problems as those NetSimulator facing, we propose the concept of *physical refactoring*, which is a disciplined technique for restructuring the physical organization of a software system, to improve the efficiency of team development and the maintainability of the system, while preserving the system's external behavior. On the one hand, physical refactoring continuously redesigns the physical structure to accommodate it to the changing situation of software development, whose redesign does not discard the existing structure, but based on it. On the other hand, physical refactoring always utilizes the standards and methods of physical design to optimize the physical structure. In one word, physical refactoring enables developers to go over the software development, maintenance and quality from the viewpoint of the physical structure at any stage of the software life cycle.

Physical refactoring is not only a simple extension of refactoring and physical design, but a powerful complement. As a refactoring technique, it benefits from the basic ideas of refactoring, including small step, frequent test, etc[2−5]. As a technique of physical design, it adopts the techniques related to physical refactoring, including design principles, methods, etc[6,7]. In this paper, we summarize these practices that many researchers have generalized, and then introduce the main techniques of physical design into the basic framework of refactoring to resolve the problems mentioned in the previous section.

3.2 Process of physical refactoring

Based on the features of refactoring and physical design, we suggest that physical refactoring should be performed as an iterative process: "identify - refactor - assess". (i) The physical structure that needs be refactored is identified depending on the developers'experiences, physical design standards, etc. (ii) A proper refactoring is chosen to adjust the special undesirable structure. (iii) The refactored system is assessed to judge whether the applied refactoring is correct and efficient. In the remainder of this section, we will discuss this process in detail.

(i) Identifying where to apply refactoring

Before refactoring, the physical structure should be measured to help developers identify where to apply physical refactorings. Good structure usually satisfies some physical design principles, rules, conventions, etc. In this paper, we mainly make use of the principles in Table 1[6,7] to guide physical refactoring, which include coupling principles and cohesion principles. The coupling primarily principles help developers evaluate compile-time dependencies, and the cohesion principles mainly assist developers to divide components into packages. Thereby, these principles not only provide the criteria to measure the structure quality, but also offer the refactoring goals that the refactored system should achieve.

Table 1 Common physical design principles

Category	Name	Descriptions
Coupling principles	Acyclic dependencies principle (ADP)	Allow no cycles in the compile-time dependency graph.
	Stable dependencies principle (SDP)	Depend in the direction of stability.
	Stable abstractions principle (SAP)	A physical entity should be as abstract as it is stable.
	Low-Build cost principle (LCP)	Cost caused by the modification of a physical entity should be low.
Cohesion principles	Reuse-Release equivalence principle (REP)	The granule of reuse is the granule of release.
	Common closure principle (CCP)	Components within a released package should share common closure.
	Common reuse principle (CRP)	The components in a package are reused together.

(ii) Refactoring

After identifying where to apply refactoring, a proper refactoring should be selected by developers in terms of the measurement results in step (i). In practical systems, violating those principles in Table 1 usually appears as unreasonable packages and undesirable compile-time dependencies. Accordingly, the main strategy of the physical refactorings is to reorganize the packages and adjust the compile-time dependencies.

Table 2 Common physical refactorings

Category	Name	Alias	Description	Towards
Adjust component dependency	Move member	Move field, move method	Move the non-cohesive part from one component into another one.	ADP, SDP, SAP, LCP
	Extract component	Extract class	Move the non-cohesive part from one component into a new one.	ADP, SDP, SAP, LCP
	Extract common component	Escalation, demotion, manager class	Extract the cohesive parts from multiple components into a new component.	ADP, LCP
	Build abstract component	Protocol Class, Extract Interface, Dependencies Inversion	Extract a service interface, and let the classes that need the service send messages to the interface directly.	ADP, SDP, SAP, LCP
	Extract object factory	Object factory	Create a special class for creating objects.	ADP, SDP, SAP, LCP
Compose component implementation	Remove private inheritance		Use embedded classes instead of private inheritance.	LCP, SDP
	Apply PImpl idiom	PImpl idiom, fully insulating, concrete class	Define a new class in the dot-C file to encapsulate the private members, declare a pointer to an object of the new class in the header file, and let the accessing to the private members by using the pointer.	LCP, SDP
	Provide header file of forward declarations		Provide header files of forward declarations for components, such as ⟨iosfwd⟩ in STL.	LCP
	Encapsulate system dependency		Provide an independent header file for each compiling system or environment.	LCP
	Move component	Move class	Move the non-cohesive component from one package into another one.	REP, CCP, CRP, ADP, SDP, SAP, LCP
	Extract package		Extract the cohesive components into a new package.	REP, CCP, CRP, ADP, SDP, SAP, LCP
	Extract abstract package		Extract the abstract components into a new package.	SDP, SAP
	Extract common package		Extract the cohesive components from multiple packages into a new one.	ADP, LCP

The common physical refactorings (shown in Table 2) are divided into three categories: *Adjust Component*

Dependency, *Compose Component Implementation* and *Organize Package*. *Adjust Component Dependency* and *Compose Component Implementation* aim at components, whose goals are to reduce the build cost of components, while *Organize Package* adjusts the package structure to decrease the build cost of packages and facilitate the parallel work of developers. In the next section, we will use NetSimulator to detail these physical refactorings.

(iii) Assessing the refactoring quality

After refactoring, the refactored system needs to be evaluated to guarantee the quality of the applied refactoring. Above all, the refactored system is tested to confirm that the refactoring does not change the external behavior of the system. Furthermore, the system is measured to make sure that the refactoring indeed improves the physical structure quality. The methods described in step (i) can also be used here to evaluate the refactored system. By comparing the measurement results before and after refactoring, it is able to judge whether the refactoring is effective. Finally, according to the information about the testing and measurement, a decision is made by developers: should physical refactoring be stopped, cancelled or continued?

4 Methods of Physical Refactoring

4.1 Adjust component dependency

Adjust Component Dependency move function points (they are expressed by semantic-related codes and provide a relatively complete service) resulting in undesirable design into other components or to build an abstract interface for such points, which makes the components satisfy ADP, SDP, SAP and LCP. In the practical process, proper refactorings should be chosen in terms of specific cases. Generally, if a component is non-cohesive, the "splitting" refactorings should be adopted, such as *Move Member*[2], *Extract Component*[3], *Extract Object Factory*[9], *Extract Common Component*[6], etc. Contrariwise, *Build Abstract Component*[2,6,7] is usually applied.

For NetSimulator in Fig.1, the cyclic dependencies in the subsystem Node and Line violate ADP and LCP. Since the undesirable dependencies are produced by creation function, we choose *Extract Object Factory* to move this function point into other components (the refactored system and codes are respectively shown in Fig.2 and Fig.3). Herein, the implementation of NFac (LFac) adopts Singleton[8] and Object Factory pattern[9], so that the dependency is reverse compared with the ordinary case. After *Extract Object Factory*, NFac and LFac are encapsulated by a new package PFac with *Extract Package* for higher cohesion and easier maintenance. Now, the subsystem fulfils ADP and LCP, thereby reducing the incremental build cost when it is modified (Extract Package will be discussed in Section 4.3).

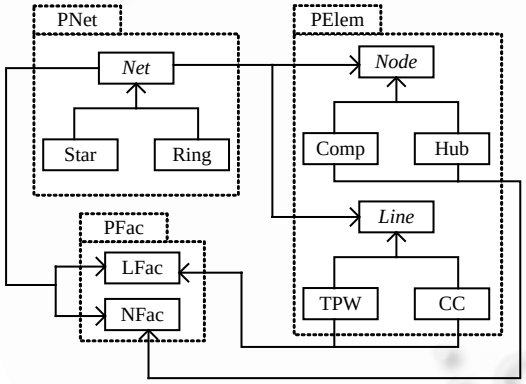


Fig.2 Extract object factory in Fig.1

4.2 Compose component implementation

Compose Component Implementation mainly removes compile-time dependencies on the private part of components to reduce the strength of these dependencies, which makes the components satisfy SAP as well as LCP. H.Sutter says: "*C++ makes private members inaccessible, but not invisible*"[10], so any modification of the header file of a component, even the private members, will cause its clients to be recompiled. For reducing such cost, this sort of refactorings is usually used. Besides, some refactorings of this sort can remove redundant dependencies on

header files, such as *Provide Header File of Forward Declarations*[11] and *Encapsulate System Dependency*[12]. In this paper, we centralize on how to remove the dependencies on the private part.

The main action *of Compose Component Implementation* is to move a component's private members from its header file to its dot-C file, so that its clients will not be disturbed by the modification of these members. This enhances the changeability of the header file. There are mainly two types of *Compose Component Implementation*. One is Partial Insulation Techniques[6], which insulates part of the private part, like *Remove Private Inheritance*[10]. The other is Total Insulation Techniques[6], which insulates the whole private part, like *Apply PImpl Idiom*[6,10,11].

Shown by Fig.2, all the other packages in NetSimulator depend on the package PFac, thus any change of its header files will result in the recompiling of the whole system. To solve this problem, we adopt *Apply PImpl Idiom* to remove the dependency on the private members of the components in PFac (the refactored codes are shown in Fig.4). After refactoring, NFac's (LFac's) private members are all defined in its dot-C file, whose modification will not conduce to the recompiling of its clients. Furthermore, this refactoring helps to ensure strong guarantee of exception safety and increase information hiding[10].

Before Refactoring

```
// In file Node.h
class Node{
public:
   // Create an object of Node's subtype
   static Node *Create(const char *typeId);
   // Other members
};
```

After Refactoring

```
// In file NFac.h
#include <map>
#include <string>
// NFac is a Singleton Object Factory
class NFac {
public:
   typedef    Node* (*CreateNode) ( );
   // Add (typeId, Creation function) to map_
   bool Register (const std::string &typeId,
                  CreateNode Fn);
   // Create concrete object based on typeId
   Node* Create (const std::string &typeId);
   // Return the handle of the singleton object
   static NFac& Only();
private:
   std::map<std::string, CreateNode> map_;
   …
};

// In file Hub.cpp
// Function to create a Hub's object
static Node* Hub::CreateHub() { return new Hub; }
const bool registered =
   NFac::Only().Register("Hub", Hub::CreateHub);
```

Fig.3 Extract object factory in Fig.1

Before Refactoring

```
// In file NFac.h
#include <map>
#include <string>
class NFac {
public:
   // public members
private:
   std::map<std::string, CreateNode> map_;
};
```

After Refactoring

```
// In file NFac.h
class NFac {
public:
   // public members
private:
   class Impl;
   Impl* pImpl_;
};

// In file NFac.cpp
#include <map>
#include <string>
#include "NFac.h"
class NFac::Impl {
   // The implementation of NFac can
   // be changed at will without
   // recompiling its clients
   std::map<std::string, CreateNode> map_;
   …
};
```

Fig.4 Apply PImpl idiom in Fig.3

4.3 Organize package

Organize Package always moves components causing defective design into other packages. The key idea here is to redistribute components across the package hierarchy to improve the package structure. Different refactorings are chosen according to cases. On the condition that a package holds both concrete and abstract components, *Extract Abstract Package* is often applied to making the package satisfy SAP, whose core is similar to *Build*

Abstract Component. In other cases, *Move Component*[13], *Extract Package*[13] or *Extract Common Package* is usually selected for enhancing package cohesion.

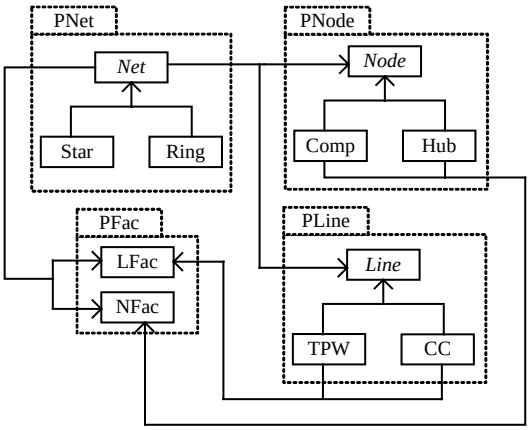


Fig.5 Extract package in Fig.3

In the rest of this section, we will continue refactoring NetSimulator to enable the team to work independently. After two refactorings, there are still some problems in NetSimulator. Above all, it violates the cohesion principles, since the subsystem Node and Line are encapsulated in the same package. In this case, Tom and Jim are unable to work independently. To overcome this problem, we adopt *Extract Package* to partition the package PElem into two cohesive packages PLine and PNode. The refactored system is depicted in Fig.5. After refactoring, Tom and Jim are able to work independently.

However, the package PNode and PLine including the implementation details violate SAP, which leads Mary to depend on Tom and Jim to a very large extent. In light of PNode and PLine containing both abstract and concrete components, we choose *Extract Abstract Package* to encapsulate the abstract components Node and Line in a new abstract package PIElem. The refactored system is shown in Fig.6. After refactoring, the package PNet only depends on the abstract package PIElem and the stable package PFac. In this case, Mary can work independently, thereby increasing her development efficiency evidently.

Howbeit in Fig.6, the package PNet providing the system's interface not only contains the abstract component Net, but also the concrete components Star and Ring, which also disobeys SAP. Accordingly, we use *Extract Abstract Package* to refactor PNet. The refactored system is shown in Fig.7. After this refactoring, we regard that the physical structure quality meets the development requirements, and the refactoring can be stopped.

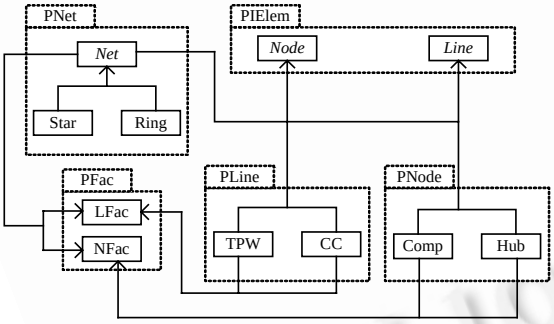


Fig.6 Extract abstract package in Fig.6

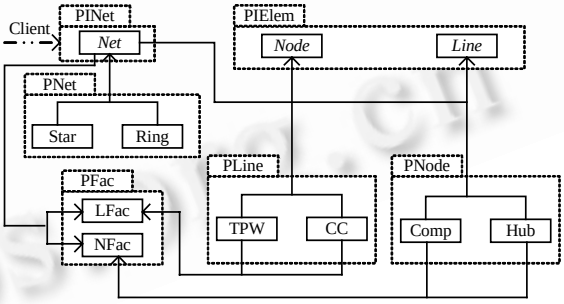


Fig.7 Extract abstract package in Fig.7

4.4 Case analysis

The refactoring example above shows that physical refactoring has ability of evidently increasing team development efficiency. On the one hand, physical refactoring makes it easier for developers to work independently. The refactorings reorganize NetSimulator's packages to make the developers not rely on the changing physical entities, which makes for working independently. Here, although the package PFac is concrete, it is very stable in this system. Thus, the dependency on it is as safe as the dependency on an abstract package. On the other hand, physical refactoring decreases build cost, which helps to coordinate the developers' work. The refactorings remove

the cyclic dependencies and insulate the private members of the factories, which reduces the complexity of compile-time dependencies. It is widely accepted that the complexity of compile-time dependencies determines build cost greatly[6].

In addition, the example indicates that physical refactoring follows the general refactoring process. First of all, before and after refactoring, the system needs assessment. Although we use our experiences to estimate NetSimulator with a view to its simplicity, the large-scale software needs metrics (tools) that can assess the structure quality quickly and exactly. Besides, physical refactoring carries on an iterative process. In general, the good physical structure cannot be obtained by only one refactoring.

5 Related Work

The methods and process of physical refactoring discussed above make it clear that physical refactoring possesses a broad practical foundation, and its basic idea is very credible. Furthermore, they also indicate that physical refactoring is not a simple superposition of physical design and refactoring.

5.1 Physical refactoring and existing refactoring

Although physical refactoring is a refactoring technique and follows the basic refactoring framework, by comparing it with the existing refactoring techniques, it has the following differences:

• **Goals**. The existing refactoring techniques are calculated for improving the reusability, understandability and changeability of a system, whereas physical refactoring aims at enhancing team development efficiency.

• **Methods**. The existing refactorings modify the logical structure, such as classes, methods, call relationship and so on. But the physical refactorings improve the physical structure by adjusting the organization of packages and the compile-time dependencies. However, there are some refactorings that can improve the quality of both the logical and physical structure, such as *Extract Object Factory* in Section 4.2.

• **Evaluation Way**. The existing refactoring techniques mainly rely on the evaluation standards of logical design, such as *Duplicated Codes* and *Long Method*[2]. Howbeit the evaluation of physical structure requires a suite of different standards, such as SAP.

Of course, the logical structure and physical structure are like two sides of a coin, so there are also some relations between physical refactoring and the existing refactoring. Firstly, applying the existing refactorings is an important reason leading to applying the physical refactorings, since the logical structure is the foundation of the physical structure. Secondly, the physical refactorings for component always change the logical structure. Thirdly, physical refactoring decreases build cost and improves testability, which accelerates the iterative process of the existing refactorings[2].

5.2 Researches of physical design

Many researchers have seriously explored physical design, whose studies primarily involve the design methods, quality measurement, tools, etc. All these researches play an important role in physical refactoring.

The design methods offer a suite of physical refactorings for enhancing the physical structure. Lakos sketches an approach for removing cyclic dependencies and introduces several compiler insulation techniques[6]. Additionally, Martin illustrates how to organize packages according to the design principles[7]. Besides, Stroupstrup, Sutter, and Meyers also present some methods to adjust compile-time dependencies[11,12, 14,15].

The quality measurement contains standards and methods. The standards are comprised of design principles, rules, guidances, conventions, etc., which give the goals, reasons and effect of physical refactoring. The methods chiefly use metrics to estimate whether the physical structure satisfies the standards. Lakos[6] and Martin[7] propose a series of design standards and metrics that support these standards. Mcconnell[16] and Evans[17] emphasizes the

importance of domain knowledge, programming standards, regulates and conventions in packages design.

There are many tools that can analyze physical design to improve the efficiency of the quality measurement. Some standard and public tools[6,18] are able to extract physical dependencies. In addition, Lakos introduces a tool to analyze physical dependencies[6]. Moreover, in terms of Lakos and Martin, Paton and Hautus, *et al*. respectively develop PDCHECK[19] and OptimalAdvisor[20] to check the physical structure.

To sum up, the techniques of physical design provide a relatively complete support for the whole refactoring process, and lay the foundation for physical refactoring as well as its further study. In this paper, we summarize these existing techniques in the round, and utilize them in the appropriate phase of the refactoring process.

6 Conclusions

In the large-scale software development, the physical structure affects team development efficiency greatly. To control its quality in software evolution, we generalize and develop the best practices that gain from refactoring and physical design to propose the concept of physical refactoring, and then dig into its process and methods. Physical refactoring is able to preserve the software quality from the viewpoint of both logical and physical design, since it has the following advantages:

• Physical refactoring extends the scope of refactoring. Compared to the existing refactoring techniques, physical refactoring centers round the physical structure that is not usually involved in logical design. Therefore, our work is an effective extension to refactoring.

• Physical refactoring preserves the quality of the physical structure in software evolution. Compared to physical design, physical refactoring allows developers to improve the physical structure at any stage of the software life cycle. Using it, a good physical design is not a precondition of development any more, but a result. Hence, our work is a powerful complement to the existing physical design techniques.

Of course, like the existing refactoring techniques, physical refactoring is a technique centering on developers in its nature. Any decision made in physical refactoring requires the judgments of developers who synthetically allow for the practical situation, various attributes of software, physical and logical structure, refactorings, etc. In this paper, we diffusely summarize the related techniques, and hope to give some helpful references to developers in the large-scale software development. In future work, we will explore the following two folds in depth:

1) The automatic support for physical refactoring. Up to now, the assessment of the physical structure is mainly conducted by manual review, which is inefficient, even infeasible, in the case of large-scale software. Besides, a lack of tools for applying physical refactorings (semi-)automatically not only decreases development efficiency, but also easily introduces bugs.

2) The relationship among physical structure, software development environment and middleware architecture. Software development environment and middleware architecture has great impact on software architecture. Since the physical structure is involved in the software architecture, it has a close relationship with the software development environment and middleware architecture.

References:

[1] Hohmann L. Beyond Software Architecture: Creating and Sustaining Winning Solutions. Addison Wesley, 2003.

[2] Flower M. Refactoring: Improving the Design of Existing Code. Addison Wesley, 1999.

[3] Mens T, Tourwé T. A survey of software refactoring. IEEE Trans. on Software Engineering, 2004,30(2):126−139.

[4] Opdyke WF. Refactoring object oriented framworks [Ph.D. Thesis]. Urbana-Champaign: University of Illinois, 1992.

[5] Kerievsky J. Refactoring to Patterns. Addison Wesley, 2004.

[6] Lakos J. Large-Scale C++ Software Design. Addison Wesley, 1996.

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

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

[9] Alexandresc A. Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, 2001.

[10] Sutter H, Alexandrescu A. C++ Coding Standards: 101 Rules, Guidelines, and Best Practices. Addison Wesley, 2004.

[11] Sutter H. Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions. Addison Wesley, 1999.

[12] Kernighan BW, Pike R. The Practice of Programming. Addison Wesley, 1999.

[13] Flower M. http://www.refactoring.com, 2007.

[14] Stroupstrup B. The Design and Evolution of C++. Addison Wesley, 1994.

[15] Meyers S. Effective C++: 55 Specific Ways to Improve Your Programs and Designs 3e. Addison-Wesley Professional, 2005.

[16] McConnell S. Code Complete 2e. Microsoft Press, 2004.

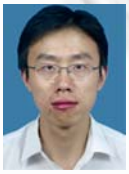
[17] Evans E. Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison Wesley/Pearson, 2004.

[18] gmake. 2007. http://www.gnu.org/software/make/

[19] Paton K. Extraction and examination of relations in C++-principles of good physical design, courtesy of lakos and martin. Dr. Dobb's Portal, 2001,26(10):28−34.

[20] Hautus E. Improving Java software through package structure analysis. In: Proc. of the 6th IASTED Int'l Conf. on Software Engineering and Applications, 2005.

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



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

XU Bao-Wen was born in 1961. He is a professor, 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 Department of Computer Science and Technology, Nanjing University. His research areas are software metrics, software evolution, and program understanding and analysis.