Вісник Харківського національного університету
Серія «Математичне моделювання. Інформаційні технології. Автоматизовані системи управління»
УДК 681.3.068 № 703, 2005, с.118-126

One Way to Guarantee the Stable Behaviour of a Software System by Preventing Unauthorized External Intrusions

G. M. Zholtkevych, I. T. Zaretska
V. N. Karazin Kharkiv National University, Ukraine

A new software design pattern of architectural level called Chest is introduced in the paper. It allows guaranteeing the stable behaviour of a complex system by encapsulating its manager into a protecting shell class with static interface. Clients can use system functions only via Chest class which delegates real work to the system manager class. Full description of this pattern is presented in the paper. The description includes pattern structure, relationship between participants, problems of realization, and examples of code and cryptosystem application.

Introduction

The problem of good design and architectural solutions reuse is in the centre of software engineering research. Successive reuse of such solutions (including the programming code) is one of the key factors in providing software development efficiency and software product quality. In fact reuse ensuring in the process of software development was one of the uppermost goals when the object-oriented paradigm of analysis, design and programming was being created [1 – 3]. As practice shows the object oriented approach to the decomposition of knowledge domain (object oriented decomposition) in contrast to the functional decomposition makes it possible to recognize invariants of a structure and behaviour which are called software patterns [4]. As founders of the object-oriented paradigm stated determining the typical interactions between objects of the designed system can be regarded as one of its quality metrics. If this aspect is given focus on the design stage the architecture of the system gets more compact, easy and flexible than if patterns were ignored (see for example the foreword to [4]). This paper describes one software pattern of architectural level, which has been discovered by authors during their work in the TEMPUS TACIS MP JEP 23010-2003 project. This pattern allows developer to protect his software subsystem from access by any methods except for those allowed by him. Such solution guarantees stable behaviour of a system, which is defined by its implementation. It is especially important for the subsystems critical in view of their information security or in view of their effect on the system environment.

Problem description and main ideas

Let us consider some subsystem with definite number of functions and a manager to control its performance in accordance to definite logic and rules. For example computer aided manufacture systems or systems of information exchange crypto protection are of such kind. Exactly the stability of system behaviour is the crucial factor for such systems. It is important to find an architectural solution to protect your system

and its manager from external impact as well as to prevent any changes in the predefined behaviour of executable components of the system. One of such architectural solutions is offered in this paper. Its main idea is to prohibit any direct access to system functions and to its manager behaviour from external objects by encapsulating them into some "protective shell" which strictly regulates user rights. This solution can be regarded as a software pattern of system or architectural level. According to its purpose we call it Chest or Coffer.

Pattern description

We will use the standard scheme to describe the pattern [4].

Pattern name and group: **Chest,** system or architectural group.

Purpose: to guarantee security and stability of complex system performance.

Also known as: **Coffer**.

Motivation. Let us consider some system for cryptographic security of information exchange between hosts in a global network. The system itself has a number of functions using definite algorithms and interacting in definite way to secure information. Certainly it is not advisable and moreover quite dangerous to permit any user an immediate access to these system functions. It would not only aggravate his work but could cause breaking information integrity or secrecy. A natural solution to this problem is to introduce a manager class, which would control the complex process of interaction between system functions. But if any user is permitted an immediate access to the manager class he could try to crack the system by extending (inheriting) or just replacing the manager class. Certainly it is possible to restrict user access to the manager class only by "allowed" methods but still the problem of manager instances creating and destroying remains unsolved. To impose this responsibility on a user (i.e. to make public the manager class constructor and destructor) means to give a user an opportunity to decide independently when and which instance of the manager class to create. Such an "excessive freedom" does not contribute to the reliability of the system performance.

The proposed solution eliminates the mentioned above disadvantages due to a new class **CryptoSystem** that works like a protecting shell. It encapsulates the instance of the manager class so it is possible to create or destroy this instance or to execute its methods only from inside this **CryptoSystem** class. No user can directly access either the system functions or its manager but interacts with the cryptosystem only via allowed for him interface of the **CryptoSystem** class (fig. 1 at the next page). Such architecture deprives user any possibility to intrude into the system work.

Usability. This pattern is used if

- it is necessary to prohibit the direct access of a user to the system functions;
- the system functions interact in a complicated way while a user needs only simple standard interface to the system;
- the responsibility to control the system functions interaction is assigned to a separate manager class that should be protected from changing or replacing.

Structure.

Participants (fig. 2 at the next page)

Client – a user of the system who:

- interacts with the system only via the class **Chest** interface;

− has restricted access to the system like starting or stopping the system or some other standard actions;

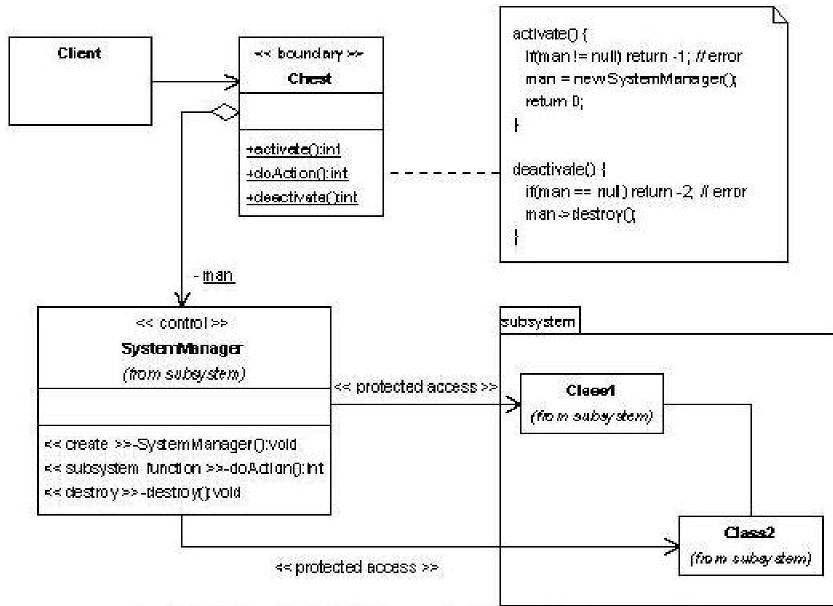

Fig. 1. Cryptosystem structure.

− knows neither about the internal structure of the system nor about the logic of its functions interaction.

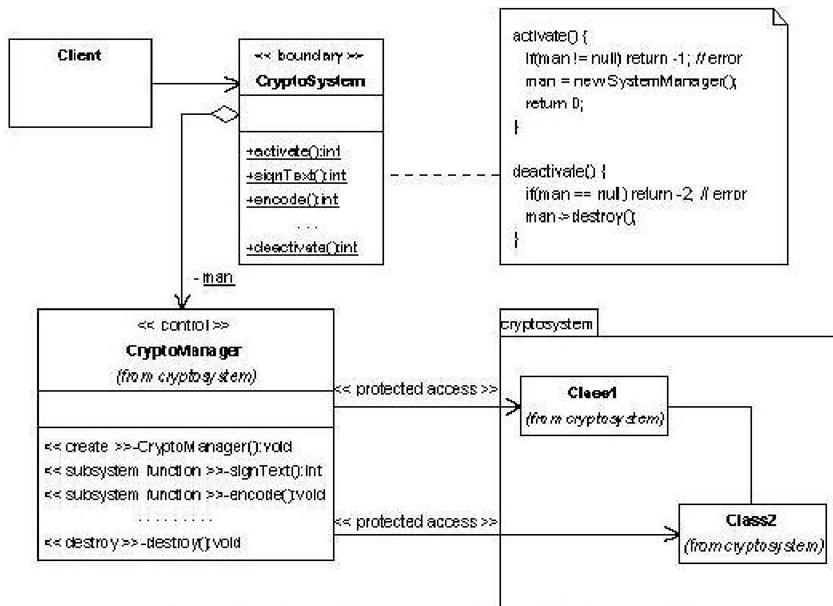

Fig. 2. Pattern Chest structure

Chest – the protecting shell which:

− encapsulates the instance of the class **SystemManager**;

- is responsible for creating the instance of the class **SystemManager** on system start and for destroying it on system stop;
- delegates the responsibility to control the system performance to the instance of the class **SystemManager**;

SystemManager – the manager of the system which:

- is closed for access from outside except for the class **Chest**;
- knows the logic of the system functions interaction and controls it;
- has an immediate access to the functions of the system which are all closed for outside access.

Subsystem – the system which:

- consists of a number of objects of interacting classes;
- has tools to provide the full functionality of the system;
- is closed for access from outside except for the class **SystemManager**.

Relations

- A user starts the system by calling the static method of the class **Chest**.
- The class Chest creates the instance of the class **SystemManager**, makes some initializing operations if necessary and delegates all control over the system work to this instance.
- The instance of the class **SystemManager** guarantees the system performance according to the logic and rules known only to this class.
- When the system stops working the class **Chest** destroys the instance of the class **SystemManager**.
- If a user needs to interrupt the system performance he calls the static method of the class **Chest** that correctly releases resources if necessary and destroys the instance of the class **SystemManager**.

Remark. The class **Chest** might have expanded interface implemented by static methods and allowing a user to vary the behaviour of the system within definite limits.

Advantages and disadvantages of the pattern **Chest**:

- *all user operations with the system are under the complete control*. Due to the encapsulation of the manager class inside the class **Chest** a user is disabled not only to call the manager class methods but also to create and destroy its instances as well. The interface of the class **Chest** allows user only limited number of operations with the system that cannot destabilize its work. Classes of the system have private interfaces so they are closed for the user access. They can be used only via the system manager. Thus the main requirement of the object-oriented approach, which is encapsulation, is strongly kept;
- *interaction between a user and the system becomes easier.* The interface of the class **Chest** is designed in such a way to supply user by the standard operations with the system. A user has no idea about the internal structure of the system as well as about the logic or implementation of these standard operations on the system level. There is even no need for him to be bothered by creating and destroying the instance of the class **Chest** since its entire interface is static. For a user working with the system is like pressing the buttons on the panel of the microwave oven to cook the meal. He could know nothing about internal structure of the oven and all the more about the properties of physical processes inside it.

- *there could be no creating of redundant objects*. Without the class **Chest** a user would be compelled to interact with the instance of the manager class to make operations with the system. In this case only him would be responsible for creating and destroying such instances, which could lead to creating redundant objects or destroying non-existing ones. With the class **Chest** one instance of the manager class is created at the beginning of the user session with the system and is destroyed at the end of the current session;
- *the manager class and the system classes are simplified*. Since a user has no access to these classes there is no need to handle user errors in their code.
- *the productivity of the system lessens a little.* To reach the required reliability of the system you have "to pay" by introducing a new class with the responsibilities of the protecting shell. This in its turn leads to an additional level of responsibility delegating and code expanding. Nevertheless neither first nor second considerably affects the system performance as the class **Chest** is quite compact and requires no instances to work with. As to the additional method calls they are necessary only for a small number of standard operations with the system from a user side.

Implementation problems

It is advisable to think over the following problems concerned with the implementation of the pattern **Chest**.

1. *How to guarantee the access to the system manager class and to the system functions only via the* **Chest** *class and to prohibit such access by any other way?*

The implementation of such access depends on the programming language. Usually object-oriented languages have flexible tools for object access rights control. Say C++ allows friend classes or functions to access private and protected data of the class while Java uses packages for this purpose. So in C++ you can declare private or protected all data and methods of the system classes and of the class manager including constructors and destructors. To control the system performance you should declare its manager class or some of its methods as a friend ones to the system classes. As to the shell class **Chest** it should be declared as a friend class to the system manager to be able to delegate to it the responsibilities for the system performance control. In Java to implement the required access you can use the visibility inside the package and default access modifier for data and methods that should be closed from outside. In this case the functionality of the package can be accessed only through the class **Chest** since it is the only class with the public interface.

2. *What is the return value of the class* **Chest** *methods?*

It is quite common situation when a user would like to know the result of his operation with the system. So it would be better to have the diagnostics of the operation results in the class **Chest**, say by the return values of the interface methods. They could be the values of some enumeration or just string messages. Certainly they should be generated by the system functions and then passed to the system manager, which would return them to the class **Chest** as a result of the delegated responsibility.

Example of code

We consider here some programming code fragments of the pattern **Chest** implementation for the user interaction with the microwave oven. Certainly we are not go-

ing to discuss complex internal mechanisms of this device but restrict ourselves only by details essential for the pattern **Chest**.

C++ code is given below.

```
// enumeration to diagnose the operation result
typedef enum Result {OK, ERROR, FATAL_ERROR};

// enumeration to chose the operating mode
// default mode on start is REHEAT
typedef enum Option {REHEAT, DEFROST, AUTOCOOK};

class MicroWaveChest;

class MicroWaveManager
{
friend class MicroWaveChest;
private:
// methods to control the oven
  Result doStart(int time); // calling the methods
                //of the MicroWave class to start
                //the oven for the given time
  Result doStop(); // calling the methods
          //of the MicroWave class to stop the oven
  Result doSetOption(Option option); //calling
      //the methods of the MicroWave class to set
      //and indicate an operating mode
  displayResult(Result result); //calling the methods
      // of the class  MicroWave to display results
      //constructor, destructor
  MicroWaveManager();
  ~MicroWaveManager();
  // data to describe the manager state
  . . . . . . . . . . . . . . . . . . . . . . . . .
};

class MicroWave
{
friend class MicroWaveManager;
private:
// data to describe the structure and the current
// state of the oven

  . . . . . . . . . . . . . . . . . . . . . . . . .

// methods to describe working mechanisms

  . . . . . . . . . . . . . . . . . . . . . . . . .
};
```

```
class MicroWaveChest
{
public:
  static void activate(){
    if (!man) man = new MicroWaveManager();}
  static Result start(int time);
  static Result stop();
  static Result setOption(Option option);
  static void deactivate() { if (man) delete man;};
private:
  static MicroWaveManager *man = NULL;
};

Result MicroWaveChest::start(int time)
{
  Result result;
  if (!man) result = FATAL_ERROR;
  else result = man -> doStart(time);
  man -> displayResult(result);
  return result;
}

Result MicroWaveChest::stop()
{
  Result result;
  if (!man) result = ERROR;
  else result = man -> doStop();
  man -> displayResult(result);
  return result;
}
Result MicroWaveChest::SetOption(Option option)
{
  Result result;
  if (!man) result = FATAL_ERROR;
  else result = man ->doSetOption(option);
  man -> displayResult(result);
  return result;
}
```

Here are some possible scenarios of user operations with the oven.

1. Simple reheating for some time, say for 1 minute:
```
MicroWaveChest::activate();
MicroWaveChest::start(60);
MicroWaveChest::deactivate();
```
2. Setting the operating mode, say to defrosting, before starting:
```
MicroWaveChest::activate();
MicroWaveChest::setOption(DEFROST);
MicroWaveChest::start(20);
MicroWaveChest::deactivate();
```

3. Stopping the oven before the timeout:
```
MicroWaveChest::stop();
MicroWaveChest::deactivate();
```

Related patterns

To design the system manager which main function is to control the system classes interaction the pattern **Mediator** can be used. To design the interaction between the class **Chest** and the manager class one can use the **Singleton** pattern and elements of the **Memento** pattern. The interaction between the **Chest** class and the system classes via the manager class can be considered in terms of the pattern **Facade** or protecting **Proxy** with slightly changed conditions for the access to real subjects. The manager class can use the pattern **State** to control the system behaviour.

Known applications

This pattern has been used in the process of the cryptographic security system development within the TEMPUS TACIS MP JEP 23010 – 2003 "UnIT – Net in universities management" project.

To complete its description we give here the interface of the class **Chest**. In this application its name is **Cryptosystem.**

```
class Cryptosystem
{
public:
   static Result activate();

   // this method defines the document file for
   // further processing
   static Result setActiveText(const char* location);

   // this mehod forms signed posting on the base
   // of the document or adds the signature to
   // the posting
   static Result signDocument();

   // this method checks the validness
   // of the signatures in the posting
   //and returns the list of invalid signatures
   static char* validateSignature();

   // this methods restores the document by its
   // posting
   static Result restoreDocument();

   // this method encodes the posting
   static Result encodePosting();

   // this method decodes the posting
   static Result decodePosting();
```

```
   // this method saves the active document in a form
   // which corresponds its current state
   static Result saveActiveText(const char* location);

   static Result deactivate();
};
```

The detailed description of the issues concerned with the implementation of the cryptographic security system within the UnIT-Net network can be found here: http://www.unit-net.org.ua

REFERENCES

1. Pascoe G. A. Encapsulators: A new software paradigm in Smalltalk-80 / Object-Oriented Programming Systems, Languages and Applications Conference Proceedings. – Portland: ACM Press, 1986. – Pp. 341 – 346.
2. Rumbaugh J., Blaha M. and others. Object-Oriented Modeling and Design. – Englewood Cliffs, NJ: Prentice Hall, 1991. – 347 p.
3. Booch G. Object-Oriented Analysis and Design with Applications. Second Edition. – Redwood City, CA: Benjamin/Cummings, 1994. – 753 p.
4. Gamma E., Helm R. and others. Design Patterns. Elements of Reusable Object-Oriented Software. – Addison-Wesley, 2003. –321 p.