AC 2007-1138: A NAFP PROJECT: USE OF OBJECT ORIENTED METHODOLOGIES AND DESIGN PATTERNS TO REFACTOR SOFTWARE DESIGN

Gholam Ali Shaykhian, NASA

Gholam "Ali" Shaykhian Gholam Ali Shaykhian is a software engineer with the National Aeronautics and Space Administration (NASA), Kennedy Space Center (KSC), Engineering Directorate. He is a National Administrator Fellowship Program (NAFP) fellow and served his fellowships at Bethune Cookman College in Daytona Beach, Florida. Ali is currently pursing a Ph.D. in Operations Research at Florida Institute of Technology. He has received a Master of Science (M.S.) degree in Computer Systems from University of Central Florida in 1985 and a second M.S. degree in Operations Research from the same university in 1997. His research interests include object-oriented methodologies, design patterns, software safety, and genetic and optimization algorithms. He teaches graduate courses in Computer Information Systems at Florida Institute of Technology's University College. Mr. Shaykhian is a senior member of the Institute of Electrical and Electronics Engineering (IEEE) and is the Vice-Chair (2005-2007), Education Chair (2003-2007) and Awards Chair of the IEEE Canaveral section. He is a professional member of the American Society for Engineering Education (ASEE), serving as the Program Chair and Web Master for the Minorities in Engineering Division of ASEE (2006-2008). He was an assistant professor and coordinator of the Information Systems program at the University of Central Florida prior to his full time appointment at NASA KSC.

Rhoda Baggs, Florida Institute of Technology

Dr. Rhoda Baggs Dr. Rhoda Baggs is the Program Chair for the MS in Computer Information Systems for Florida Institute of Technology's University College. This program specializes in object-oriented programming, component engineering, data driven systems, and other related software, system, and IS topics. She has earned a Ph.D. and an M.S. in Computer Science from the Florida Institute of Technology and a Bachelor of Science in Computer Science from the University of Pittsburgh. In between and during academic achievements, Dr. Baggs has worked primarily as a Software Engineer for such companies as Texas Instruments, Raytheon, JDS Uniphase, Optical Process Automation, WT Automation, Advanced Manufacturing Technologies, Inc., and NASA. Research Interests include Multimedia Tutorials and Software Engineering for the Same, Software Engineering and Reverse Engineering of Legacy Software, Image Processing, and Systems Engineering. Dr. Baggs is a member of the International Association of Computer Information Systems, the Association of Computing Machinery, and several ACM SIGs, including Design of Communication, Information Technology Education, Software Engineering, and Multimedia.

© American Society for Engineering Education, 2007

An NAFP Project: Use of Object Oriented Methodologies and Design Patterns to Refactor Software Design

Gholam Ali Shaykhian, NAFP Fellow
Engineering Directorate
National Aeronautics and Space Administration
Kennedy Space Center
Ali.Shaykhian@nasa.gov

Rhoda Baggs, Ph.D.
Assistant Professor of Computer Sciences
University College
Florida Institute of Technology
rbaggs@fit.edu

Introduction

In the early problem-solution era of software programming, functional decompositions were mainly used to design and implement software solutions. In functional decompositions, functions and data are introduced as two separate entities during the design phase, and are followed as such in the implementation phase. Functional decompositions make use of refactoring through optimizing the algorithms, grouping similar functionalities into common reusable functions, and using abstract representations of data where possible; all these are done during the implementation phase. This paper advocates the usage of object-oriented methodologies and design patterns as the centerpieces of refactoring software solutions. Refactoring software is a method of changing software design while explicitly preserving its external functionalities. The combined usage of object-oriented methodologies and design patterns to refactor should also benefit the overall software life cycle cost with improved software.

Object-oriented methodology

The software development discipline is undergoing a tremendous metamorphosis, brought on by the influence of new development paradigms, new development tools, new technologies, more complex requirements, and ever-shorter development cycles. The usage of object-oriented methodology in constructing engineering and business applications has grown exponentially since the early 90's. In the object-oriented methodology, the software design focuses on objects instead of functions and functional decompositions. An object is introduced as a discrete entity, containing its data and functions. The main aspects of the object-oriented methodology includes encapsulation, inheritance and polymorphism [3]. Encapsulation refers to wrapping object attributes and behaviors in an enclosed entity, inheritance deals with object reuse, and polymorphism concerns with object having access to a behavior where the knowledge to the access is known at runtime.

Objects encapsulate the related attributes (data or member data) and behaviors (functions or member functions) of an entity. In practice, an ill design of an object is to wrap a set of unrelated data and functions enclosed in a named entity; hence making it difficult to refactor. Representation of an object should provide tight internal coupling of the object's data and functions and loose coupling of the object's external usage. Design of an object, should be required to encompass *only* related data and functions of that object. Explicit definition of an object in this form lends itself to significant software reuse. When internal members (data and functions) of an object are tightly coupled, changes to a member's data are only possible through its corresponding member functions. The external usage of the object should be loosely coupled to the object and a client should not directly change the object data. Instead the request to change the object's data is sent to the object via the object's member functions via messaging. If a client wants to change an object's data, it sends a message to the object, requesting for the change.

Relationships among objects are similar to those known to us in real life [3]. Suppose an object of type Student has registered for a course with an instructor. A set of attributes and behavior describe the student and professor objects. Professor evaluates his/her student's paper and exam. A professor is a member of a college/department within the university. A university has several colleges/departments. A professor object, college/department, and university have roles and responsibilities in processing a student's grade. Object relationships formalize the relationship among objects, these relationships are *knows-a*, *is-a*, *has-a*, and *depends-a*. Additional relationships among objects can be derived from these base relationships: for example *with-a*.

The *knows-a* relationship describes an association between two objects; for example, a student knows his/her professor through registering for a course. This knowledge can be unidirectional (a professor knows a student) or bi-directional (both the student and the professor know each other). The *has-a* relationship is when an object is composed of other objects. Professors are members of their college/department (college/department

has professors) and a University has colleges/departments. These are examples of the *has-a* relationship. The *is-a* relationship describes inheritance. In our example student *is-a* person and also professor *is-a* person. When there is the need to establish relationships among objects that deal with limited privileges, these limited privileges can be modeled as *depends-a* relationships. For example, a college may want to restrict the creation of new student objects and only allow certain instructors to have privilege of creating new student objects. Object dependency can be used to regulate restrictions among objects. An object can defer binding to its member's functions to run time. This behavior is known as polymorphism, and is implemented via dynamic binding or late binding. Dynamic binding eliminates the implementation of the look up table when similar functionality is required.

Object Oriented Methodology and Design Patterns to Refactor Software Design

An intrinsic property of software in a real-world environment is its need to evolve. Software evolution concerns every phase of the software life cycle: the requirements phase through the maintenance phase. The traditional software life cycle includes phases for software requirements, implementation, testing and maintenance. Software evolution may involve 1) introducing new behavior in which case it is considered a maintenance activity; 2) modifying and extending the existing software design behavior in which case ill posed requirements may be attributed to the formulation of the software requirements; or 3) restructuring the software design to improve quality factors such as readability or improved designed in which case it is considered software refactoring. Refactoring using design patterns is one of the promising approaches to improve the designs during development activities, and a crucial issue is to identify when, where and which patterns could be applied [4]. Refactoring is a disciplined technique for restructuring an existing design or body of code, altering its internal structure without changing its external behavior. That is to say, refactoring is a technique to improve the maintainability of software. By definition, refactoring is the transformations of code and design specifications while explicitly preserving its unique design and functionalities.

Refactoring tends to permit and reveal numerous opportunities to improve the software. The term "refactoring" in software engineering, means modifying design or source code without changing its external behavior, with the motivation being to improve software maintenance cost. Refactor to understand is a typical reverse engineering pattern in that it does not intend to improve the code base or the design itself, but improves the maintainers understanding. Consequently, less emphasis is put on regression testing, and more on the composition and verification of hypotheses concerning the code. The iterative process of *Refactor to Understand* [5] is described as 1) Read the code, 2) Evaluate names on their correspondence to the true semantics of the variable/method/class and rename if necessary, and 3) Evaluate groups of statements on their semantic coherence [5]. The use of design patterns to implement refactoring is a promising approach to improve conceptually the productivity of the software

development process and thus to reduce both the cost and time of developing and maintaining complex systems.

An example scenario

Earlier in this paper, Student, Professor, College/Department, University, and Course objects were used to explain the relationships among objects. This scenario explores refactoring through the use of design patterns. Suppose an instructor wishes to keep a list of all graduate students who have taken their courses and have earned an "A". This particular professor requires all his/her research assistants to maintain an "A" average to be considered for the assistantship. Also, each department wishes to keep track of graduate students that have maintained an average of "B" or better. Furthermore, a student can only work as either a graduate teaching assistant or research assistant but not both. Students are required to check with their professors or their respective departments for potential opportunities for teaching or research assistantships. Figure –1 shows the object relationships among Student, Professor, Department, College and University. Each department maintains a list of faculties and list of students and tracks those students that are awarded assistantships.

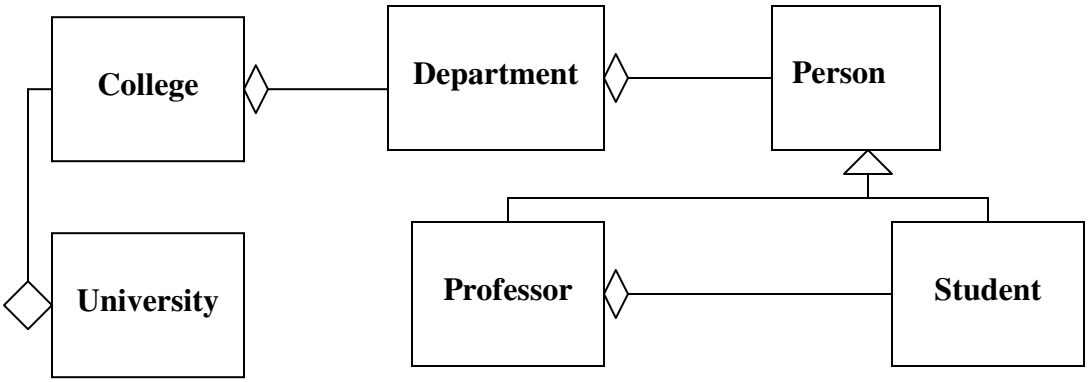

Figure 1: Object Relationships

Figure-1 uses diagrams which are defined [7] in the Unified Modeling Language (UML), an object modeling and specification language. UML provides a rich set of diagrams to represent objects and object relationships. Here only diagrams used in this paper are described. A class diagram, represented by a rectangle (a box), is used to describe objects with common structure and common behavior. Object relationships show the communication path between two objects, the communication paths are knows-a, has-a, is-a and depends-a relationships. Within UML the knows-a relationship is represented by a line with an optional arrowhead indicating the role of the object(s) in the relationship, the has-a relationship is represented by a line connecting two related classes with a diamond next to the class representing the whole, the is-a relationship is represented by a line connecting two related classes with a triangle next to the super class (parent class, base class), and dependency (depends-a) is represented by a dashed line and exists

between two defined elements if a change to the definition of one would result in a change to the other.

The presented design (Figure-1) uses object-oriented methodologies as the basis for the software solution. When an object-oriented solution is done correctly, the maintenance cost of it is less than an equivalent functional solution mostly due to the coupling and the cohesion factors described below:

- Object data and member functions are encapsulated as one entity.
- Object data are hidden (private member) from the client.
- The accesses to object data are limited to its member functions.
- A client needing object data makes requests through public member functions.
- Changes to object data are possible within the object.

Patterns are a recent software engineering problem-solving discipline that has emerged from the object-oriented methodologies. A pattern is the abstraction for describing recurring solutions to common problems in software design [1]. The notion of design patterns is to build a body of knowledge to support the design and development. Crafting design patterns during the design phase will allow programs to share knowledge about their design and is the basis for a recurring solution. More specifically, the concrete form which recurs is that of a solution to a recurring problem. The origin of design patterns lies in work done by an architect named Christopher Alexander during the late 1970s. Patterns have roots in many disciplines, including literate programming, and most notably in Alexander's work on urban planning and building architecture [1].

In real world scenarios, problems occur within a certain context, and in the presence of numerous competing solutions. The design analysis and refinement phase propose solutions in the manner that is most appropriate for the given context. Design patterns are identified with a unique name, for example, “Abstract Factory”, “Subject-Observer”, or “Singleton” and a description. The description of the pattern tries to capture the essential insight which it embodies, so that others may learn from it, hence providing recurring solutions to common problems.

Use of design patterns to refactor the design also improves the data processing of a software system, for example:

- Each professor keeps a list of his/her assistants.
- Additional coordination at the college/department level is required to disallow multiple assistantships to a student.
- Students are required to regularly check with their faculties for research assistantships.
- Students are required to check with their department/college for teaching assistantships.

The "Abstract Factory" and "Subject-Observer" design patterns [1] are well suited candidates to offer an enhanced design for this problem and to demonstrate the use of refactoring of the software design.

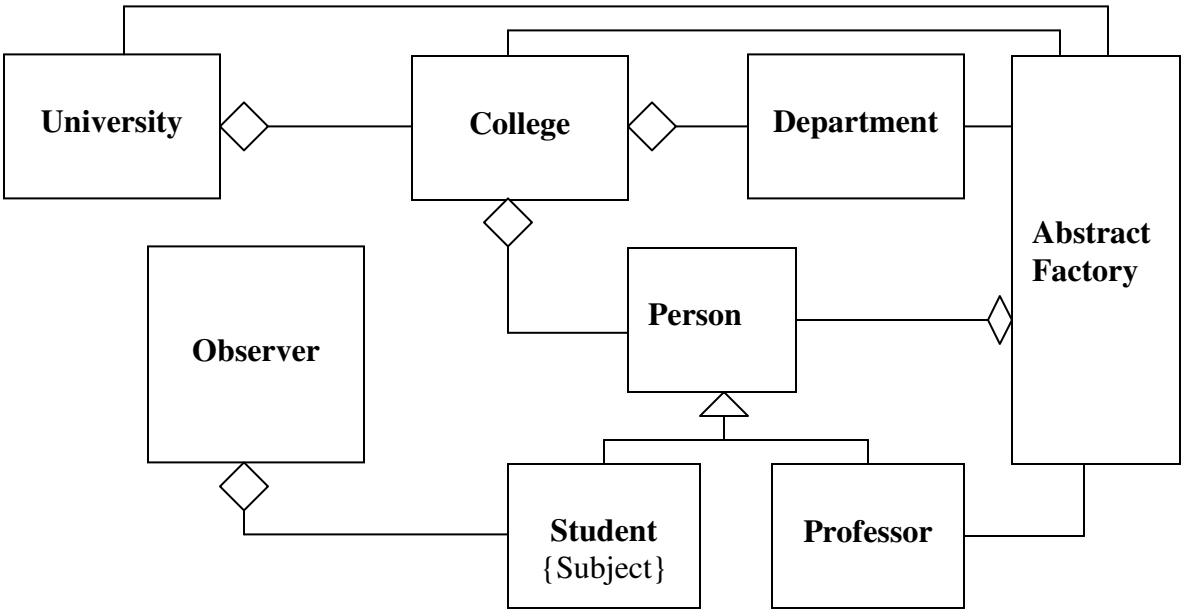

Figure 2: Use of Design Patterns

In Figure-2, Student and Professor inherit from Person class (is-a relationship), University has Colleges (has-a relationship), College has Department (has-a relationship), University, College and Department are associated with the Abstract Factory (knows-a relationship).

The intent of the Observer pattern is to define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. For example, all students will receive notifications of assistantships when it becomes available eliminating the need for students to regularly check with college/instructors for the availability of assistantships. The Abstract Factory provides an interface for creating families of related or dependent objects without specifying their concrete classes [1].

Design depicted in Figure-2 consolidates object creations since all students and professor objects are created in the Abstract Factory class. Student objects register their interest with the Observer class to receive notifications of when assistantships become available. The notification message may include professor and department/college information offering the assistantship, hence eliminating the need for students to constantly check

with their instructors or department/college of availability of assistantship opportunities. Furthermore, the use of Abstract Factory eliminates the need for each professor to keep a list of his/her assistants or requiring additional coordination at the college/department level since all objects are only created in factory.

```
/*
        Abstract Factory -- Intent: Provide an interface for creating families of related or
        dependent objects without specifying their concrete classes

        Singleton -- Intent: Ensure a class only has one instance, and provide a global
        point of access to it.
*/

#include <iostream>
#include "Student.h"
#include "Professor.h"

class Person;
class PersonFactory{
public:
static PersonFactory *instance()
{
       if(!PF) {
          PF = new PersonFactory(); }
       return PF;
}
Student * createStudentObject()
{
      return new Student();
}

Person * createPerssorObject()
{
     return new Professor();
}

protected:
     PersonFactory(){}
private:
     static PersonFactory *PF;
};
```

Table 1 – Abstract Factory C++ code

Table-1 shows a partial C++ [2] implementation code for the Abstract Factory design pattern. Student and instructor objects are created in the factory. Attempt to create objects elsewhere will result in a compile time error. Regulating this violation through design

helps reduce software maintenance costs since it is not required to manually inspect all code to make sure no violation is made.

```
/*
        Observer -- Intent: Define a one-to-many dependency between objects so that
        when one object changes state, all its dependents are notified and updated
        automatically

        Singleton -- Intent: Ensure a class only has one instance, and provide a global
        point of access to it.
*/

#include <iostream>
#include <list>
#include <iterator>
#include <algorithm>

using namespace std;
class Person;
class Observer {
public:

static SubStation * instance()
{
   if(!SS)  {
      SS = new SubStation; //singleton object, only executed once
   }
   return SS;
}

void registerStudent(Person * objectPtr)
{
   RegisteredObjectList.push_back(objectPtr);
}

void notify()
{
    list<Person *>::iterator i;
    for(i = RegisteredObjectList.begin(); i != RegisteredObjectList.end(); i++)
    {
        (*i)->noticeAssistantship(message);
    }
}
private:
  string  message;
  static Observer *SS;
  // RegisteredObjectList list of all dependent objects When one object changes
```

```
   // state, all its dependents are notified and updated automatically.
   static list<Person *> RegisteredObjectList;
};
```

Table 2 – Observer C++ code

The Observer design pattern in Table 2 keeps a list of all students who are qualified for assistantships in the `RegisteredObjectList` container object. The

```
(*i)->noticeAssistantship(message)
```

line in the notify member function sends a message to all student objects, notifying them of the availability of an assistantship.

Summary

In this paper, the focus is on refactoring software through design patterns. This is achieved by first defining objects and relationships among objects. The goal of applying design patterns to refactor software design is not to present entirely novel solutions to problems, but to disseminate good solutions known to experts and to provide a vocabulary for talking about these solutions. Design patterns have proven valuable for bringing design reuse to object-oriented programming, for establishing common practices and for providing a vocabulary among scientists, engineers and educators.

Bibliography

1. Gamma, A. Helm, R. Johnson, R. and Vlissides, J., Design Patterns, Elements of Reusable Object-Oriented Software, New York: Addison-Wesley, 1995.
2. Stroustrup, B., The C++ Programming Language, New York: Addison-Wesley, 2000.
3. Shaykhian, G.A., Implementation of Business policies using object-oriented methodologies and design patterns. Proceedings of the 2005 American Society for Engineering Education Annual Conference & Exposition.
4. Muraki, T. and Saeki, M., Metrics for Applying GOF Design Patterns in Refactoring Processes. International Workshop on Principles of Software Evolution (IWPSE), Vienna, Austria, 27-36, 2001.
5. Bois, B.D., Demeyer, S. and Verelst, J., Does the ”Refactor to Understand” Reverse Engineering Pattern Improve Program Comprehension? Proceedings of the Ninth European Conference on Software Maintenance and Reengineering (CSMR’05), IEEE Computer Society, 1534-5351 (2005).
6. Buckley, J., Mens, T., Zenger, M., A. Rashid and G. Kniesel, Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice, Published online in Wiley InterScience (www.interscience.wiley.com), 17:309–332 (2005).
7. Blaha, M. and Rumbaugh J., Object-Oriented Modeling and Design with UML, 2/E, Prentice Hall, 2004.

Gholam "Ali" Shaykhian
Gholam Ali Shaykhian is a software engineer with the National Aeronautics and Space Administration (NASA), Kennedy Space Center (KSC), Engineering Directorate. He is a National Administrator Fellowship Program (NAFP) fellow and served his fellowships at Bethune Cookman College in Daytona Beach, Florida. Ali is currently pursing a Ph.D. in Operations Research at Florida Institute of Technology. He has received a Master of Science (M.S.) degree in Computer Systems from University of Central Florida in 1985 and a second M.S. degree in Operations Research from the same university in 1997. His research interests include object-oriented methodologies, design patterns, software safety, and genetic and optimization algorithms. He teaches graduate courses in Computer Information Systems at Florida Institute of Technology's University College. Mr. Shaykhian is a senior member of the Institute of Electrical and Electronics Engineering (IEEE) and is the Vice-Chair (2005-2007), Education Chair (2003-2007) and Awards Chair of the IEEE Canaveral section. He is a professional member of the American Society for Engineering Education (ASEE), serving as the Program Chair and Web Master for the Minorities in Engineering Division of ASEE (2006-2008). He was an assistant professor and coordinator of the Information Systems program at the University of Central Florida prior to his full time appointment at NASA KSC.

Dr. Rhoda Baggs
Dr. Rhoda Baggs is the Program Chair for the MS in Computer Information Systems for Florida Institute of Technology's University College. This program specializes in object-oriented programming, component engineering, data driven systems, and other related software, system, and IS topics. She has earned a Ph.D. and an M.S. in Computer Science from the Florida Institute of Technology and a Bachelor of Science in Computer Science from the University of Pittsburgh. In between and during academic achievements, Dr. Baggs has worked primarily as a Software Engineer for such companies as Texas Instruments, Raytheon, JDS Uniphase, Optical Process Automation, WT Automation, Advanced Manufacturing Technologies, Inc., and NASA. Research Interests include Multimedia Tutorials and Software Engineering for the Same, Software Engineering and Reverse Engineering of Legacy Software, Image Processing, and Systems Engineering. Dr. Baggs is a member of the International Association of Computer Information Systems, the Association of Computing Machinery, and several ACM SIGs, including Design of Communication, Information Technology Education, Software Engineering, and Multimedia.