Introductory Software Engineering with a Focus on Dependency Management

Christine Hofmeister
East Stroudsburg University, East Stroudsburg, PA, USA
chofmeister@esu.edu

Abstract. In this paper I describe my approach to teaching introductory software engineering, a three-credit upper-level undergraduate course. What I believe is innovative is not the particular concepts or techniques but the way they are woven together, using the unifying concept of dependency management. The specific techniques I teach are the use of interfaces and factory design patterns. Interfaces are used to control call-dependencies, and by using C++ (which has no native interface), students learn precisely how an interface differs from a class. Factories are used to control creation dependencies, by enabling object creation while hiding the type of object being created. I advocate a design focus in this course because that helps ground the concepts closer to the code base. UML models and class specifications are used to abstract the code. The UML models summarize the relationships found in the code and help students reason about dependencies.

Keywords: software engineering curriculum, interfaces, design patterns, dependency management, UML modeling.

1 Introduction

This paper describes an approach to teaching introductory software engineering with a focus on dependency management. I have evolved this approach over the course of 14 years of teaching a senior-level (4th year) undergraduate course of 3 credit hours (3 hours of instruction per week for 14 weeks).

I have adopted this dependency management focus as a result of my seven years working in industry. I saw first-hand how the judicious use of interfaces had a significant benefit in a family of software systems, and that students were not learning this as part of their undergraduate education. In this same family of systems, my colleagues and I used the factory design pattern to solve the problem of needing to instantiate an object without knowing at compile-time what the object type would be. While the factory pattern will never be needed as frequently as interfaces, it nonetheless solves an otherwise intractable dependency problem. My fourteen years of teaching these two concepts (interfaces and factory design patterns) in the undergraduate software engineering course have convinced me that they are an excellent vehicle for teaching students practical techniques for dependency management.

Another thing I have observed is that students who have not had any experience in large-scale software development do not readily grasp software engineering concepts. To these students the concepts come across as platitudes, since they don't have the depth of experience to really understand them. This problem has been observed also by others [8]. To address this problem I teach a variety of specific techniques, grounded where possible at the code level, because undergraduates have the most experience at this level. One example is the use of UML diagrams to model C++ code. Another example is determining whether a given implementation obeys a class specification (given as pre and post conditions on the operations).

The use of code-level models has the important benefit of precisely revealing various kinds of dependencies. Interfaces of course allow us to directly control dependencies, and this property is revealed in a UML Class Diagram. In a UML Sequence Diagram, however, the interfaces dissolve. So the use of UML diagrams at the code level nicely supports the teaching of interfaces. The factory design patterns extend dependency management into the realm of object creation. The factory patterns allow us to create an object without knowing (exactly) which type of concrete object is being created. I use two variants, "Class Factory" and "Factory Method," which differ in the mechanism for supplying the concrete object type.

The reasons for using C++ in this course are that it remains a prevalent, mainstream language, and that it is a rich language with multiple ways of accomplishing things. Variables can be allocated statically (at compile-time) or dynamically (using "new"), and there is no garbage collection. These are good vehicles for teaching object memory allocation. Its lack of native interface support means that in order to write an interface in C++, students learn precisely the difference between a class and an interface, and even about constructor and destructor chaining. Its support of both static and dynamic method binding is good for understanding polymorphism. Finally, since it supports both methods and classes, it can be used to teach libraries and linking, and enables us to use the factory method design pattern.

I would expect a typical introductory-level software engineering course to include some UML and to advocate the use of interfaces. The course described in this paper differs in that UML and interfaces receive a deeper treatment and are taught as techniques for managing dependencies. The dependency focus also encompasses the teaching of the factory design patterns for managing creation dependencies.

The next three sections of the paper describe these three key aspects of the course: modeling and specification, interfaces, and the factory design patterns. These are followed by sections on course evaluation, related work, and a concluding summary.

2 Modeling and Specification

In addition to being used to model the software at various levels of abstraction, UML is often used to model the problem domain. However, I found that most students were confused by seeing multiple uses of UML, so now I teach only how to use it to model code. This has the added benefit that there is no ambiguity in determining whether a diagram is correct of not, which has obvious pedagogical advantages.

UML Class Diagram. The software engineering students use UML Class Diagrams for modeling a set of related classes and interfaces. To translate C++ code into a class diagram, the students follow a set of conventions, most of which are the same as those used by a typical round-trip engineering UML tool such as Rational Rose.

The conventions describe when attributes should be shown in the attribute compartment or drawn as a composition or aggregation relationship to another class in the diagram. The basic dependency relationship is used for local variables, parameters, and method invocations. For object creation we use the dependency relationship with a <<creates>> stereotype. The standard inheritance (generalization) and implements (realization) relationships are used, as is the standard interface notation. We use a class symbol with the stereotype <<method>> to show how a static method is related to classes, e.g. with the main() method. (Since we use C++, methods need not be encapsulated in a class.) We do not typically show library classes such as string.

In addition to learning the language of the UML Class Diagram, students learn what C++'s "virtual," "pure virtual," and "static" mean in more general object-oriented terminology. They are able to see visually how an interface is a barrier for dependencies (see Figure 1).

UML Sequence Diagram. UML sequence diagrams require few additional conventions for modeling C++ code, beyond using a <<method>> stereotype for class methods (known as static methods in C++). In the sequence diagrams, interfaces dissolve and inheritance is flattened. In other words, while the class diagrams show an object's static type (the type seen by the compiler), the sequence diagram shows its dynamic type (the type it was created as).

Another thing the sequence diagrams reveal is the fact that C++ implicitly invokes the constructor when variables are declared statically as opposed to dynamically:

```
Stack s;  // compile-time allocation
Stack* sp = new Stack();  // run-time allocation
```

This experience of modeling C++ code in a sequence diagram also deepens the students' understanding of object-oriented languages.

Specification of Class Behavior. To describe class behavior the students are given a specification consisting of pre- and post-conditions for each method in the class. From this specification I ask them to draw a State Chart Diagram describing the class's interaction protocol, and to determine whether an implementation fulfills the specification. The interaction protocol (i.p.) is the allowed sequences of method invocations.

Of the UML modeling we do, our use of the State Chart Diagram is the most challenging for students, since it involves first extracting the i.p. then modeling it. The i.p. is implied by the specification; it is not necessarily explicit in the pre- and post-conditions. The approach is to use the state chart to model just one class, with the possible invocations of this class's methods as transitions between states. The constructor call labels the transition from the start state to the first state, and the destructor labels all transitions to the end state. If a caller does not obey the interaction protocol, the behavior of the class is unspecified; in those cases the code could do anything or nothing.

The second use of these class specifications is to give the students a description of an implementation and ask whether the implementation conforms to the specification. One example is an implementation of the FIFO queue not as a linked list but as a circular buffer. A second example is an underspecified Bank class that allows an implementation to impose a "service charge" on deposits.

The purpose of these exercises is to teach the students to think carefully about what the specification really says, without giving the specification the benefit of their domain knowledge, and without allowing the specification to dictate implementation details. These exercises reinforce the work the students do writing Requirements Specifications, by using the same specification concept but applying it at the class level (closer to the code base).

3 Interfaces

In general, interfaces present a unified, stable view of a component and eliminate call-dependencies on a concrete class. They are used to hide variants, to specify a callback (e.g. with GUI components), and to reduce the impact of future changes (below the interface).

In order to prepare the students for using interfaces in C++, we first review polymorphism and abstract classes. For polymorphism, a simple example of a parent class and two subclasses suffices to show the difference between virtual and non-virtual operations. When a method is not virtual, C++ uses static method binding: the compiler resolves the method call by using the object's static type, in this case the parent type. This runs faster, since no dynamic lookup is needed for the method call. When a method is virtual, C++ uses dynamic method binding (aka dynamic dispatch). The method call is not resolved until runtime, when the object's dynamic type is used to decide which method should be called. In this case it could be either of the subclass types. This polymorphic behavior is what is commonly found in object-oriented languages.

Because C++ does not have native interface support, we use a set of conventions for making a C++ class act as an interface. A C++ interface is an abstract class that contains only public abstract ("pure virtual") operations. No attributes are allowed, nor are private or protected operations. We also put a virtual destructor with an empty body in the interface, in order to preserve destructor chaining. (See Item 7 in [7], and see [9].) The destructor is defined inline so that no .cpp file is needed. An example of a C++ interface is shown in Table 1.

Table 1. Example of a C++ Interface (defined in IMessage.h)

```
class IMessage { public:
     virtual void setMessage(std::string) = 0;
     virtual void showMessage() = 0;
     virtual std::string readMessage() = 0;
     virtual ~IMessage() {}
};
```

Although interfaces are defined in .h files, a typical .h file is not an interface. Unfortunately it is common, even in some textbooks, to refer to the class declaration in the .h file as the class's "interface." For C++ classes, the convention is to declare the class in a .h file and define the operations (provide method bodies) in a .cpp file. The .h file is a mechanism for giving the class declaration to the compiler (via a #include). It can contain attributes, private or protected operations, and even method bodies (inline functions). None of these belong in an interface.

To motivate the use of an interface, I show the students the following very simple Hello example. It prompts the user for a first and last name, then, depending on whether the executable was built using the PLAINTEXT or SECURE option, displays the name in plaintext or encrypted. The main() method creates a Name object, then calls Name's firstName() and lastName() methods and prints the results. The Name class creates two Message objects, one for input and one for output. Methods first-Name() and lastName() use the output object to set and show the prompt, and use the input object to capture the user's input and return it to the caller.

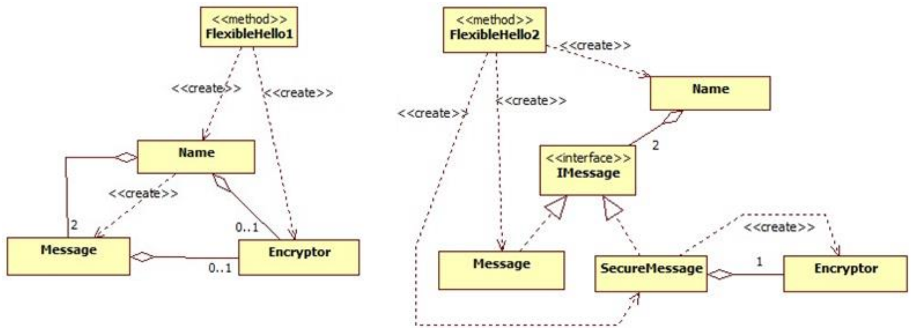

Figure 1. Hello Example: Initially (left) and with an Interface (right)

The main() method, the classes, and their relationships are shown in the UML Class Diagram in Figure 1 (left side). Notice that FlexibleHello1 (the main()) knows about all the variants of Message that exist: it creates a Name object either with an Encryptor (SECURE option) or not (PLAINTEXT option). Name does not know about the Mes-sage variants. It simply passes the Encryptor* when it constructs its Message objects. Message is capable of providing both plaintext and encrypted messages.

At this point in the class I ask the students to add a third kind of message, a LOUD (all caps) message. They see that this design is clumsy: the Encryptor* does double duty as a flag and as access to the Encryptor object, and the design is not easily exten-sible for other kinds of message objects.

Next I show the students a new design for the application, one that uses an inter-face. Instead of making the Message class handle multiple message variants, we create a separate class for each variant. These message variants are hidden behind an inter-face IMessage. This design is shown on the right in Figure 1.

Each message variant (Message, SecureMessage) is a separate class that imple-ments interface IMessage. The Encryptor is now encapsulated in SecureMessage. The main() (FlexibleHello2) still chooses which variant of message is used, but now it creates the message object and passes it as an IMessage to Name. Name depends only on interface IMessage, and knows nothing about the variants. When Name invokes setMessage(), showMessage(), or readMessage(), the method that executes depends on which type of object FlexibleHello2 created.

However, since Name needs two message objects (two instances), the main() must create both and pass both to Name, which is clumsy. The main() could even create two different kinds of messages to pass to Name, which is probably not what Name would like. A better solution would allow Name to create its own Message objects, but still keep Name ignorant of the message variants. This is the motivation for the factory design patterns.

After presenting this version, I ask the students to add the LOUD message variant to it, and compare the effort to what was needed for the initial version.

4 Factory Design Patterns

A factory design pattern helps in managing dependencies that arise due to the creation of objects. In object-oriented languages, in order to instantiate an object, you must use its class name. The only way to create an object without knowing its concrete type is to use indirection, which is how factory patterns work.

Class Factory. There are two factory design patterns I use in the software engineering course. The first I call "Class Factory," since it is a class whose methods do nothing other than create objects. This pattern is known as the 'Abstract Factory' in [6]. A class factory is a class that simply creates objects. The code for Message Factory is shown in Table 2.

Table 2. Code for Class MessageFactory

In MessageFactory.h:	In MessageFactory.cpp:

`class MessageFactory : public`
`IMessageFactory { public:`
`  MessageFactory();`
`  virtual IMessage* newMessage();`
`  virtual ~MessageFactory();`
`};`
	`MessageFactory::MessageFactory(){ }`
	`MessageFactory::~MessageFactory(){ }`
	`IMessage* MessageFacto-`
	`ry::newMessage()`
	`{ return new Message(); }`

Different class factories create different kinds of objects. Figure 2 (left) shows the FlexibleHello example revised to use a class factory. The main() decides which kind of factory object to create: with MessageFactory, IMessageFactory::newMessage() creates a Message. With SecureMessageFactory, IMessageFactory::newMessage() creates a SecureMessage.

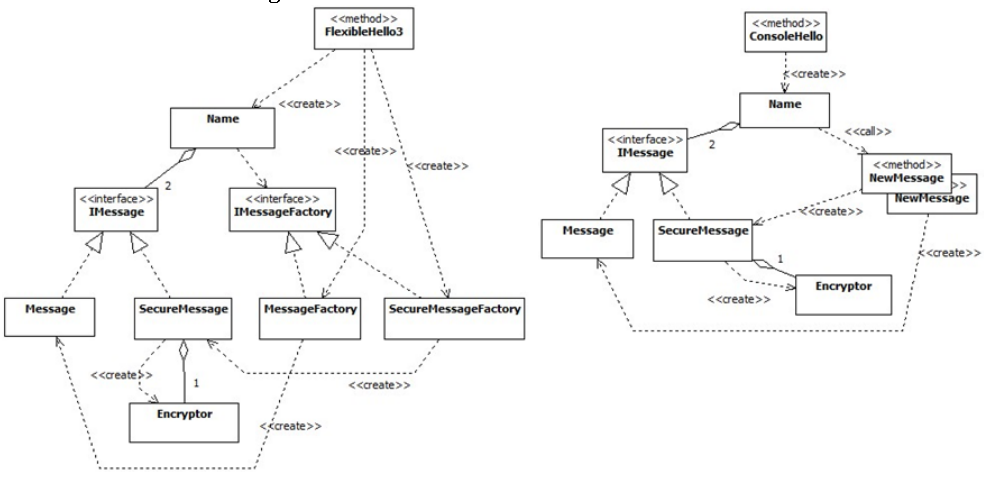

Figure 2. Hello Example with Class Factory (left) and with Factory Method (right)

Next main() passes its factory object as an IMessageFactory object to the Name object. Name does the creation of the IMessage object by calling IMessageFactory's method newMessage(). Depending on which kind of IMessageFactory object it received from main(), newMessage() creates either a Message or a SecureMessage, and returns this object to Name as an IMessage.

In summary, main() chooses (and creates) the Factory. Name uses the IMessageFactory to create an IMessage object (without knowing exactly what type of object it is creating). Name uses only the interfaces IMessageFactory and IMessage, even though the objects it accesses via these interfaces have concrete types MessageFactory & Message or SecureMessageFactory & SecureMessage.

Clearly this design is more complicated; adding a level of indirection for object creation comes at a price. However, as the students see when they add a third variant (the LoudMessage), Name now has the ability to create its own messages without knowing which variant it is creating.

Factory Method. The second factory pattern is called "Factory Method," but it is not the same as the 'Factory Method' pattern in [6], which puts the factory method in a class. The names we use are meant to help the students remember the key difference between the two patterns. This second factory pattern comes directly from my industrial experience.

The pattern we use is simpler than the class factory because instead of creating a factory class for each variant, we provide multiple definitions of a C++ static method that is not inside a class. Each variant is compiled and linked into a separate library, and at build time or runtime we choose one of the libraries to link in. Table 3 shows the method declarations and definitions for the Hello example.

Table 3. Declaration and Definition of a Factory Method

Decl. in IMessage.h	Definition in Message.lib	Def. in SecureMessage.lib
`IMessage* NewMessage();`	`IMessage* NewMessage() { return new Message(); }`	`IMessage* NewMessage() { return new SecureMessage(); }`

If a dynamic link library (aka shared library) is used, then the variant can be selected and linked in at runtime. We use static libraries in the software engineering course because libraries are new to most students and static libraries are simpler than dynamic link libraries.

In this version of the Hello example, shown in Figure 2 (right), the main() is called "ConsoleHello" rather than "FlexibleHello" because the main does not contain any code to choose variants. With static libraries the application contains no code related to selecting variants. With dynamic link libraries all that is needed at runtime is the name and path of the library file, which need not be hardcoded into the application.

The declaration for the NewMessage() method goes in IMessage.h (within the include guard, but declared *after* the class, not as part of the class). A definition of NewMessage() goes in the same .cpp file as the class it creates. For example, SecureMessage's version of NewMessage() goes in SecureMessage.cpp.

When Name needs to create an IMessage object, it simply calls NewMessage(). For an in-class exercise, as before the students add a third variant, a LoudMessage. There is little new code needed to accomplish this, but there is additional work needed for creating a library and linking it into the executable.

Combining the Factory Design Patterns. The class factory pattern moves the creation dependency to another part of the system. Normally this is used to produce a coherent set of objects, where each factory provides a set of methods newX(), newY(), etc. appropriate for that variant. See ‘Abstract Factory’ [6] for more discussion of where this is useful. The factory method pattern moves the creation dependency to a link-time binding. For maximum flexibility these two patterns can be used together: the factory method pattern is applied to a factory. The factory classes implement the interface IFactory, and the method NewFactory() has multiple definitions, one for each concrete factory class.

5 Evaluation

This course has evolved in the fourteen years I have taught it, and nearly all of the present elements have been in place for the last ten or eleven years. The changes made to arrive at the present form were primarily driven by analysis of student performance on exams and assignments, and students’ comments and questions. The two course elements described in this paper, modeling and design patterns, have been part of the course since I started teaching it. What has changed is the role of these topics in the course.

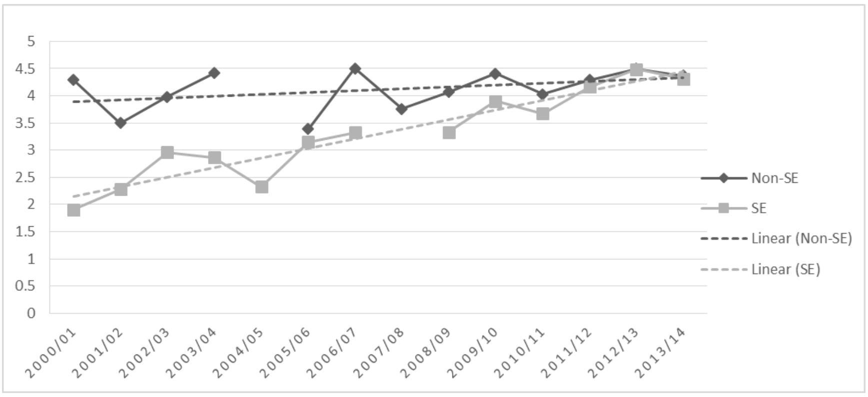

Figure 3: Students’ Evaluations of Course Quality

Students evaluate each course each semester. As can be seen in Figure 3, their ratings have improved as the course has evolved. The graph shows my students’ rating of course quality, comparing the Software Engineering (SE) course to all my other courses (Non-SE). In addition, students are asked to comment on the strongest and weakest aspects of the course, and how the course could be improved. Students find the project experience both frustrating and rewarding, as would be expected. For the strongest aspect of the course, students have noted the active learning approach, the progression of exercises (in-class examples, in-class exercises, homework, project work), and the modeling.

Other important forms of evaluation are student exit interviews (for graduating students) and regular surveys of our alumni and of employers of our students. Students value the course particularly for the practical techniques they learn. Alumni and employers regularly cite UML and design patterns as very important topics for computer science majors.

Students also tell me when a job interview touches on one of the topics we covered in the course. One of our students was asked about the difference between a class and an interface during an interview, and believes that his answer to this question clinched the job offer.

6 Related Work

A number of authors mention the problem of getting students to appreciate the challenges of software engineering. Rather than spending most of the course surveying the software engineering discipline, Rajlich advocates project development and teaching of specific skill training to occupy the bulk of the course [8]. My approach is similar.

Putting a design focus in the introductory software engineering course, as I do, is also advocated in [11] and [10]. In [3] Boehm sees COTS components and model-driven development as playing an increasing role in software development. This argues for the kind of focus on dependency management that I advocate.

In [5] Fox and Patterson describe their experiences teaching software engineering using an agile process and cloud computing. They motivate this approach in part by interviews with “a half-dozen leading software companies.” They note “the number-one request from each company: that students learn how to enhance sparsely documented legacy code.” This is completely consistent with my experience in the software industry, and has been the driving force behind my focus on modeling and dependency management.

Badreddin et. al. provide compelling evidence that UML and Umple (a textual modeling language) provide significant benefit over Java when students or software professionals are trying to understand a software system [2]. This finding supports the importance of teaching a modeling language to students. The models used in their experiments were models of source code, which is consistent with how I use UML.

Just three years after the classic design patterns book [6] was published, Astrachan et. al. argued for the inclusion of design patterns in the computer science curriculum [1]. They have become an accepted part of teaching object-oriented design, but are not always included in an introductory software engineering course, where there is already so much other material to be covered. Denzler and Gruntz advocate using design patterns and software design as a bridge between programming language courses and the first software engineering course [4].

7 Conclusion

While from the start I taught modeling (UML) and design patterns as part of the introductory Software Engineering course, over time my approach has evolved. The changes can be summarized as follows:

- Make design a more prominent aspect of the course, make dependencies the primary design consideration, and make reuse and managing variants the primary design goals.

- Narrow the focus so that models are taught in one context only (as an abstraction of source code), and the design patterns are taught via the factory patterns.
- Use modeling and design patterns to support the design goals of reuse and managing variants.
- Provide a logical progression of design solutions that starts with inheritance and polymorphism, then adds interfaces, and ends with the factory design patterns (class factory, then factory method, then combined factory patterns). Use modeling to expose the dependencies in these design solutions and to see their impact at runtime.

I am convinced these are significant improvements, and this is corroborated by the marked improvements in student satisfaction with the course, as evidenced in the course evaluation scores.

References

[1] Owen Astrachan, Garrett Mitchener, Geoffrey Berry, and Landon Cox. 1998. Design patterns: an essential component of CS curricula. In *Proceedings of the 29th SIGCSE technical symposium on Computer science education* (SIGCSE '98), D. Joyce and J. Impagliazzo (Eds.). ACM, New York, NY, USA, 153-160.

[2] Omar Badreddin, Andrew Forward, and Timothy C. Lethbridge. 2012. Model oriented programming: an empirical study of comprehension. In *Proc. of the 2012 Conf. of the Center for Advanced Studies on Collab. Research* (CASCON '12), H. Jacobsen, Y. Zou, and J. Chen (Eds.). IBM Corp., Riverton, NJ, USA, 73-86.

[3] Barry Boehm. 2006. A view of 20th and 21st century software engineering. In *Proceedings of the 28th International Conference on Software Engineering* (ICSE '06). ACM, New York, NY, USA, 12-29.

[4] Christoph Denzler and Dominik Gruntz. 2008. Design patterns: between programming and software design. In *Proceedings of the 30th international conference on Software engineering* (ICSE '08). ACM, New York, NY, USA, 801-804.

[5] Armando Fox and David Patterson. 2012. Crossing the software education chasm. *Commun. ACM* 55, 5 (May 2012), 44-49.

[6] Gamma, Helm, Johnson, Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[7] Meyers, Scott, 2007. *Effective C++, Third Edition.* Pearson Education.

[8] Václav Rajlich. 2013. Teaching developer skills in the first software engineering course. In *Proceedings of the 2013 International Conference on Software Engineering* (ICSE '13). IEEE Press, Piscataway, NJ, USA, 1109-1116.

[9] http://stackoverflow.com/questions/318064/how-do-you-declare-an-interface-in-c

[10] Taylor, R.N.; Van der Hoek, Andre, "Software Design and Architecture: The once and future focus of software engineering," *Future of Software Engineering, 2007. FOSE '07*, pp.226-243, May 2007.

[11] van Vliet, H., 2006. Reflections on software engineering education. *Software,* IEEE, vol.23, no.3, pp.55-61, May-June 2006.