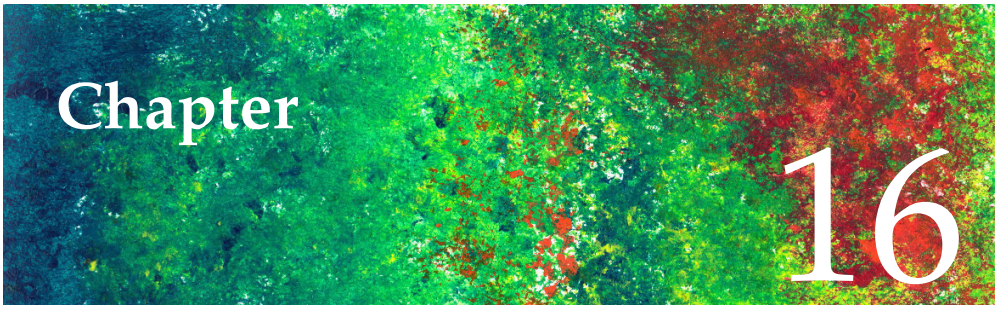


Compositional Design Principles

Learning Objectives

In this chapter, I will more formally introduce the three principles that form the ③-①-② process. The learning focus is understanding how these principles manifest themselves in our design and in our concrete implementation, and how they work in favor of increasing the maintainability and flexibility qualities in our software.

16.1 The Three Principles

The original design pattern book (Gamma et al. 1995) is organized as a catalogue of 23 design patterns. It provides, however, also an introduction that discusses various aspects of writing reusable and flexible software. In this introduction they state some principles for reusable object-oriented design.

Compositional Design Principles:

① *Program to an interface, not an implementation.*

② *Favor object composition over class inheritance.*

③ *Consider what should be variable in your design.*
(or: Encapsulate the behavior that varies.)

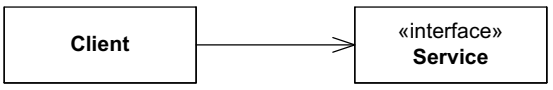
The authors themselves state the first two directly as principles (p. 18 and 20) whereas the third principle is mentioned as part of the process of selecting a design pattern (p. 29) and thus not given the status of a principle. Shalloway and Trott (2004) later highlight the statement *Consider what should be variable* as a third principle. Grand (1998) restated the two first principles as *fundamental patterns*, named INTERFACE and DELEGATION.

These three principles work together nicely and form the mindset that defines the structure and responsibility distribution that is at the core of most design patterns. You have already seen these three principles in action many times. They are the ③-①-② process that I have been using over and over again in many of the preceding chapters.

③ I identified some behavior that was likely to change…	=	③ *Consider what should be variable in your design.*
① I stated a well-defined responsibility that covers this behavior and expressed it in an interface…	=	① *Program to an interface, not an implementation.*
② Instead of implementing the behavior ourselves I delegated to an object implementing the interface…	=	② *Favor object composition over class inheritance.*

16.2 First Principle

① *Program to an interface, not an implementation.*



A central idea in modern software design is *abstraction*. There are aspects that you want to consider and a lot of details that you do not wish to be bothered with: *details are abstracted away.* We humans have limited memory capabilities and we must thus carefully select just those aspects that are relevant and not overburden ourselves with those that are not.

In object-oriented languages, the *class* is a central building block that encapsulates a lot of irrelevant implementation details while only exposing a small set of public methods. The methods abstract away the implementation details like data structures and algorithms.

Encapsulation and abstraction lead to a powerful mindset for programming: that of a *contract* between a *user* of functionality and a *provider* of functionality. I will use the term *client* for the user class which is common for descriptions of design patterns—do not confuse it with the "client" term used in internet and distributed computing. The providing class I will term the *service*. Thus there exists a contract between the client (*"I agree to call your method with the proper parameters as you have specified…"*) and the server (*"if you agree to provide the behavior you have guaranteed"*).

The question is then whether the class construct is the best way to define a contract between a server and its client. In many cases, the answer is "no!" It is better to *program to an interface* for the following reasons:

Clients are free to use *any* service provider class. If I couple the client to concrete or abstract classes I have severely delimited the set of objects that can be used by the client: an object used by the client *has* to be a subclass! Thus the client has high coupling to a specific class hierarchy.

Consider the two situations a) and b) in Figure 16.1 that reflect a development over time. In situation a) the developers have applied the *program to an interface* principle

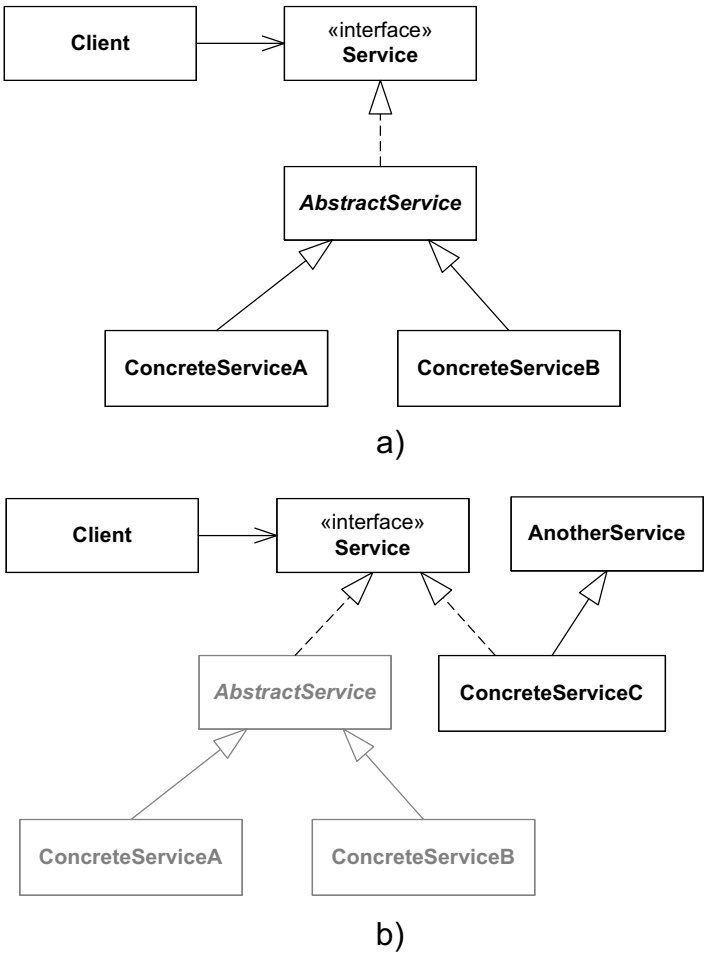


Figure 16.1: a) Original design b) Later design.

and provided a hierarchy of classes that the client uses. Later, however, it becomes beneficial to let the client collaborate with an instance of AnotherService. The interface decouples the client from the concrete implementation hierarchy.

Exercise 16.1: Describe what would happen if in situation a) the client was directly coupled to the AbstractService and you had to make the ConcreteServiceC. How would you do that?

Interfaces allow more fine-grained behavioral abstractions. By nature, a class must address all aspects of the concept that it is intended to represent. Interfaces, however, are not coupled to concepts but only to behavioral aspects. Thus they can express much more fine-grained aspects. The classic example is the Comparable interface, that only expresses the ability of an object to compare itself to another. Another example is the FigureCollection and SelectionHandler interface, explained in Section 15.7.2, that showed how introducing fine-grained abstractions can lead to reuse.

Often interfaces are used as **Private Interfaces** (Newkirk 1997) or **Role Interfaces** (Fowler 2006), that is, fine-grained and specialized "mini-roles" that allows specialized access or mutation of an object, only available to specific collaborators. I will discuss this concept in more detail later in Section 15.8.2

Interfaces better express roles. The above discussion can be rephrased in terms of the role concept. Better designs result when you think of the *role* a given server object will play for a client object. This mind set leads to making interfaces more focused and thus smaller.

Again, consider the Comparable interface in the java collection library. To the sorting algorithms, the only interesting aspect is that the objects can play the comparable role—just as a Hamlet play is only interested in a person's ability to play Hamlet. All other aspects are irrelevant.

Classes define implementation as well as interface. Imagine that the client does not program to an interface, but to a concrete service class. As a service class defines implementation, there is a risk that the client class will become coupled to its concrete behavior. The obvious example is accidentally accessing public instance variables which creates high coupling. A more subtle coupling may appear if the service implementation actually deviates from the intended contract as stated by class and method comments. Some examples are for methods to have undocumented side effects, or even have defects. In that case the client code may drift into assuming the side effects, or be coded to circumvent the defective behavior. Now the coupling has become tight between the two as another service implementation cannot be substituted.

16.3 Second Principle

> ② *Favor object composition over class inheritance.*

This statement deals with the two fundamental ways of reusing behavior in object-oriented software as outlined in Figure 16.2.

Class inheritance is the mechanism whereby I can take an existing class that has some desirable behavior and subclass it to change and add behavior. That is, I get complex behavior by reusing the behavior of the superclass. *Object composition* is, in contrast, the mechanism whereby I achieve complex behavior by *composing* the behavior of a set of objects.

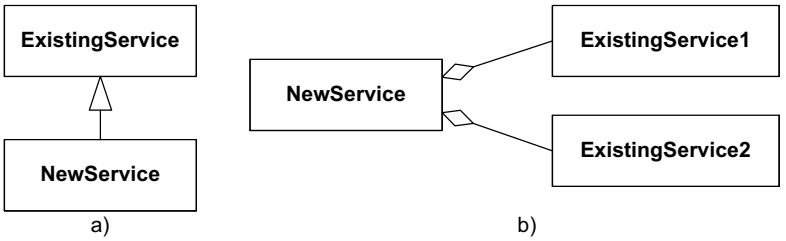


Figure 16.2: Class inheritance (a) and object composition (b).

You saw these two techniques discussed in depth in Chapter 7, *Deriving Strategy Pattern*, and in many of the following chapters concerning design patterns. The polymorphic proposal suggested using class inheritance to introduce a new rate structure algorithm; the compositional proposal suggested using object compositions to do the same.

Class inheritance has a number of advantages. It is straightforward and supported directly by the programming language. The language support ensures that you write very little extra code in order to reuse the behavior of the superclass. Below, I will describe liabilities of class inheritance, followed by a few for object composition.

Encapsulation. It is a fact that "inheritance breaks encapsulation" (Snyder 1986). A subclass has access to instance variables, data structures, and methods in all classes up the superclass chain (unless declared `private`.) Thus superclass(es) expose implementation details to be exploited in the subclass: the coupling is high indeed. This has the consequence that implementation changes in a superclass are costly as all subclasses have to be inspected, potentially refactored, and tested to avoid defects.

Object composition, in contrast, depends upon objects interacting via their interfaces and encapsulation is ensured: objects collaborating via the interfaces do not depend on instance variables and implementation details. The coupling is lower and each abstraction may be modified without affecting the others (unless the contract/interface is changed of course).

You can only add responsibilities, not remove them. Inheriting from a superclass means *"you buy the full package."* You get *all* methods and *all* data structures when you subclass, even if they are unusable or directly in conflict with the responsibilities defined by the subclass. You may override a method to do nothing in order to remove its behavior, or indicate that it is no longer a valid method to invoke, usually by throwing an exception like UnsupportedOperationException. Subclasses can only *add*, never *remove* methods and data structures inherited. A classic example is *java.util.Stack*. A stack, by definition, only supports adding and removing elements by push() and pop. However, to reuse the element storage implementation, the developers have made Stack a subclass of Vector, which is a linear list collection. That is, an instance of stack also allows elements to be inserted at a specific position, `stack.add(7, item);`, which is forbidden by a stack's contract!

Composing behavior, in contrast, leads to more fine-grained abstractions. Each abstraction can be highly focused on a single task. Thus cohesion is high as there is a clear division of responsibilities.

Exercise 16.2: Apply the ② principle to the stack example above so clients cannot invoke methods that are not part of a stack's contract but the stack abstract still reuses the vector's implementation.

Compile-time versus run-time binding. Class inheritance defines a compile-time coupling between a subclass and its superclass. Once an object of this class has been instantiated its behavior is defined once and for all throughout its lifetime. In contrast, an object that provides behavior by delegating partial behavior to delegate objects *can* change behavior over its lifetime, simply by changing the set of delegate objects it uses. For instance, you can reconfigure a Alphatown pay station to become a Betatown pay station even at run-time simply by changing what rate strategy and what factory it uses.

Exercise 16.3: Extend the pay station so it can be reconfigured at run-time by providing it with a new factory object. You will have to introduce a new method in the PayStation interface, for instance

```
public void reconfigure(PayStationFactory factory);
```

Recurring modifications in the class hierarchy. A force I have often seen in practice is that classes in a hierarchy have a tendency to be modified often, as new subclasses are added. As an example, consider a service that fulfills its contract nicely using a simple ArrayList data structure, see a) in Figure 16.3. Later I need a better performing service implementation but if I simply subclass the original service class I have to override all methods to use a better performing data structure, and instantiated objects will contain both structures. The logical consequence is to modify the class hierarchy by adding a common, abstract, class, as shown in pane b) of the figure. While the modification is sensible, it does mean that three classes are now modified

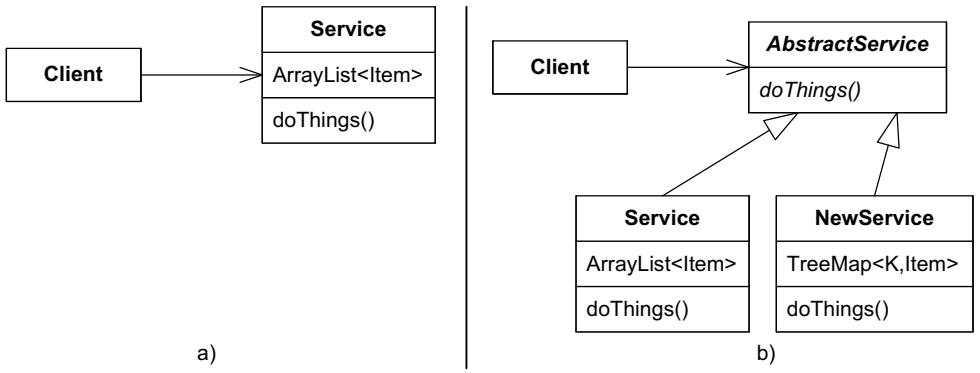


Figure 16.3: Modifications in hierarchy.

and have to be reviewed and tested to ensure the former reliability. Often, each new subclass added provides opportunities for reworking the class hierarchy and as a consequence the stability quality suffers. In Chapter 11, *Deriving State Pattern*, I discussed the tendency for subclass specific behavior "bubbling" up the hierarchy to end in the abstract superclass that becomes bigger and less cohesive over time.

In a compositional design, the array list based and tree map implementations would be separate implementations without overlap. However, this benefit may turn into a liability if the AbstractService can make concrete implementations of many of the methods. In that case, a compositional proposal would end up with duplicated code or be required to do further decomposition to avoid it. Thus, in this case one should carefully consider benefits and liabilities before committing to either solution. This particular case is explored in the exercise below.

Exercise 16.4: To make Figure 16.3 concrete, consider that the service is a list storing integers. A demonstration implementation may look like

Fragment: exercise/compositional-principles/InitialImplementation.java

```
class IntegerList {
  private int contents[]; int index;
  public IntegerList() { contents = new int[3]; index = 0;}
  public int size() { return index; }
  public boolean add(int e) {
    contents[index++] = e;
    return true;
  }
  public int get(int position) {return contents[position];}
  // following methods are data structure independent
  public boolean isEmpty() { return size() == 0; }
  public String contentsAsString() {
    String result = "[";
    for ( int i = 0; i < size()-1; i++ ) {
      result += get(i)+", ";
    }
    return result + get(size()-1)+"]";
  }
}
```

Note that the two last methods are implemented using only methods in the class' interface, thus they can be implemented once and for all in an abstract class.
Take the above source code and implement two variants of an integer list: one in which you subclass and one in which you compose behavior. Evaluate benefits and liabilities. How can you make a compositional approach that has no code duplication and does not use an abstract class?

Separate testing. Objects that handle a single task with a clearly defined responsibility may often be tested isolated from the more complex behavioral abstraction they are part of. This works in favor of higher reliability. Dependencies to *depended-on units* may be handled by test stubs. The separate testing of rate strategies, outlined in Chapter 8, is an example showing this.

Increased possibility of reuse. Small abstractions are easier to reuse as they (usually) have fewer dependencies and comes with less behavior that may not be suitable in a reusing context. The selection handler abstraction in MiniDraw, described in the previous chapter, is an example of this.

Increased number of objects, classes, and interfaces. Having two, three, or several objects doing complex behavior instead of a single object doing it all by itself naturally leads to an increase in the number of objects existing at run-time; and an increase in the number of classes and interfaces I as a developer have to overview at compile-time. If I cannot maintain this overview or I do not understand the interactions then defects will result. It is therefore vital that developers *do* have a roadmap to this web of objects and interfaces in order to overview and maintain the code. How to maintain this overview is the topic of Chapter 18.

Delegation requires more boilerplate code. A final liability is that delegation requires more "boilerplate" code. If I inherit a superclass, you only have to write `Class B extends A` and all methods are automatically available to any B object without further typing. In a compositional design, I have potentially a lot of typing to do: create an object reference to A, and type in all "reused" methods and write the delegation code:

```
void foo() { a.foo(); }
int bar() { return a.bar(); }
```

16.4 Third Principle

> ③ *Consider what should be variable in your design.*

This is the most vague of the three principles (perhaps the reason that Gamma et al. did not themselves state it as a principle). Instead of considering what might force a design change you must focus on the aspects that you want to vary—and then design your software in such a way that it can vary *without* changing the design. This is why it could be reformulated as *Encapsulate what varies in your design*: use the first two principles to express the variability as an interface, and then delegate to an object implementing it.

This principle is a recurring theme of many design patterns: some aspect is identified as the variable (like "business rule/algorithm" in STRATEGY) and the pattern provides a design that allows this aspect to vary without changes to the design but by changes in the configuration of objects collaborating.

16.5 The Principles in Action

The principles can be used by themselves but as I have pointed out throughout this book they often work nicely in concert: the ③-①-② process.

③–Consider what should be variable. I identify some behavior in an abstraction that must be variable, perhaps across product lines (Alphatown, Betatown, etc.), perhaps across computing environments (Oracle database, MySQL database, etc.), perhaps across development situations (with and without hardware sensors attached, under and outside testing control, etc.) as shown in Figure 16.4.

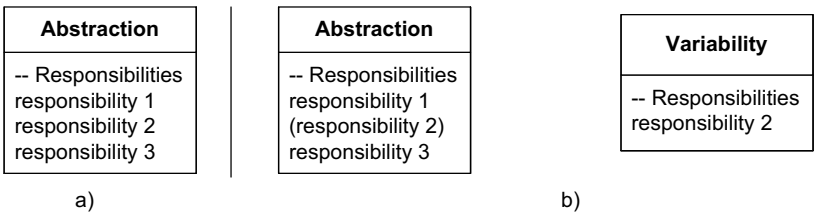


Figure 16.4: A responsibility (a) is factored out (b).

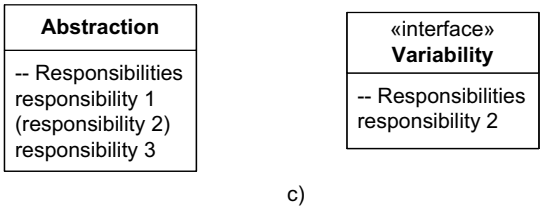


Figure 16.5: Expressing it as an interface (c).

①**–Program to an interface, not an implementation.** I express that responsibility that must be variable in a new interface, see Figure 16.5.

②**–Favor object composition over class inheritance.** And I define the full, complex, behavior by letting the client delegate behavior to the subordinate object: *let someone else do the dirty job*, as seen in Figure 16.6.

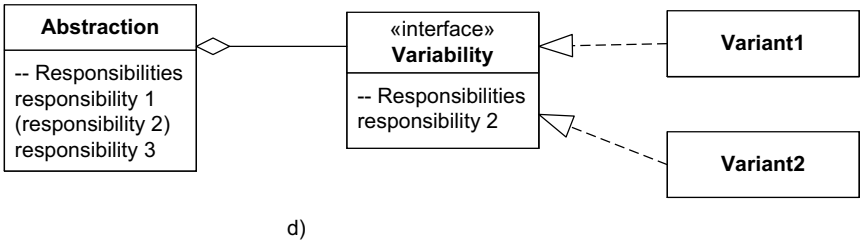


Figure 16.6: Composing full behavior by delegating (d).

Remember, however, that the ③-①-② is not a process to use mechanically. As was apparent in the discussion of the abstract factory you have to carefully evaluate your options to achieve a good design with low coupling and high cohesion. Note also that they are *principles*, not *laws*. Using these principles blindly on any problem you encounter may "over-engineer" your software. If you are not in a position where you can utilize the benefits then there is little point in applying the principles. Remember the TDD value: *Simplicity:* You should build or refactor for flexibility when need arises, not in anticipation of a need. Often when I have tried to build in flexibility in anticipation of a need, I have found myself guessing wrong and the code serving the flexibility actually gets in the way of a better solution.

16.6 SOLID

SOLID is an mnemonic acronym that is often used, which basically covers the principles stressed throughout this book. The principles were promoted by Robert C. Martin (Martin 2000) while Michael Feathers later introduced the acronym itself (Feathers 2017).

The principles are:

S The single-responsibility principle: "There should never be more than one reason for a class to change." That is, encapsulate behavior in well-defined and fine-grained roles; encapsulate what varies.

O The open–closed principle: "Software entities . . . should be open for extension, but closed for modification." That is, favor change by addition.

L The Liskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it." That is, program to an interface.

I The interface segregation principle: "Many client-specific interfaces are better than one general-purpose interface." That is, express behavior using fine-grained roles.

D The dependency inversion principle: "Depend upon abstractions, [not] concretions." That is, program to an interface, and favor object composition by dependency injection.

16.7 Summary of Key Concepts

Three principles are central for designing compositional designs. These are:

Compositional Design Principles:

① *Program to an interface, not an implementation.*

② *Favor object composition over class inheritance.*

③ *Consider what should be variable in your design.*
(or: Encapsulate the behavior that varies.)

Generally, applying these patterns makes your design more flexible and maintainable as abstractions are more loosely coupled (first principle), bindings are run-time (second principle), and abstractions tend to become smaller and more cohesive. The third principle is a keystone in many design patterns that strive to handle variability by encapsulation, using the first two principles.

16.8 Selected Solutions

Discussion of Exercise 16.1:

One possible way would be to create a subclass AbstractServiceD and let it create an instance of AbstractServiceC. All methods in AbstractServiceD (which are the ones the client invokes) are overridden to call appropriate methods in AbstractServiceC. However, the construction is odd, as AbstractServiceD of course inherits algorithms and data structures from AbstractService that are not used at all.

This proposal resembles the ADAPTER pattern, however adapter is fully compositional.

Discussion of Exercise 16.4:

You can find solutions to the exercise in folder *solution/compositional-principles*. Basically, you can do the same thing with a compositional design as with an abstract class: you factor out common code into a special role, CommonCollectionResponsibilities, implement it, and delegate from the implementations of the integer list. However, due to the delegation code and extra interfaces, the implementation becomes longer (100 lines of code versus 85) and more complex.

16.9 Review Questions

What are the three principles of flexible software design? How are they formulated? Describe and argue for their benefits and liabilities.

How do these principles relate to patterns like STRATEGY, ABSTRACT FACTORY and others that you have come across?

What are the alternative implementations that arise when these principles are not followed?

16.10 Further Exercises

Exercise 16.5:

Many introductory books on object-oriented programming demonstrate generalization/specialization hierarchies and inheritance by a classification hierarchy rooted in the concept *person* as shown in Figure 16.7. For instance a person can be a teacher or a student and by inheriting from the Person class all methods are inherited "for free": getName(), getAge(), etc.

This design may suffice for simple systems but there are several problems with this design that you should study in this exercise. You should analyze the problems stated in the context of an object-oriented university management system that handles all the university's associated persons (that is both teachers and students).

Life-cycle problem. Describe how to handle that a student graduates and gets employed as a teacher?

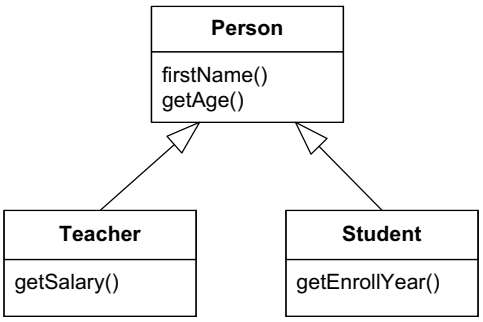


Figure 16.7: A polymorphic design for teachers and students.

Context problem. Describe how the system must handle that a teacher enrolls as student on a course? How must the above class diagram be changed in order to fully model that a person can be both a student as well as a teacher?

Consistency problem. Describe how the system must handle a situation where a teacher changes his name while enrolled in a course.

Based on your understanding of roles and by applying the principles for flexible design propose a new design that better handles these problems. Note: It is not so much the ③-①-② process that should be used here as it is the individual principles in their own right.

As a concrete step, consider the following code fragment that describes the consistency problem above:

```
Teacher t = [get teacher ``Henrik Christensen'']
Student s = [get student ``Henrik Christensen'']

assertEquals(``Henrik'', t.firstName() );
assertEquals(``Henrik'', s.firstName() );

[Person Henrik renamed to Thomas]

assertEquals(``Thomas'', t.firstName() );
assertEquals(``Thomas'', s.firstName() );
```

The point is that the name change should be a single operation at the *person* level (as name changes conceptually has nothing to do with neither Henrik's teacher nor student association).

How would you make the person-teacher-student design so that this test case passes without making changes to multiple objects?