2 Agile Design

Syllabus

What Is Agile Design ?, SRP : The Single-Responsibility Principle, OCP : The Open-Closed Principle, LSP : The Liskov Substitution Principle, DIP : The Dependency-Inversion Principle, ISP : The Interface-Segregation Principle.

Contents

2.1 What is Agile Design ?

- "What is Software Design ?" was an important article written by Jack Reeves in the C++ Journal in 1992. Reeves claims in this article that a software system's design is essentially documented through its source code. The source code diagrams are only a part of the design; they are not the design itself. Jack's article turned out to be a forerunner of agile development.
- You should not interpret this to mean a separate collection of UML diagrams from the code. A series of UML diagrams may represent elements of a design, but they are not the design itself. A design of software project is an abstract idea.
- It refers to the program's overall shape and structure, as well as the specific shape and structure of each module, class and method. It can be represented in a variety of ways, but source code is the final form. The source code is, in the end, the design.

2.1.1 What Goes Wrong When it Comes to Software ?

- If you're lucky, you'll begin a project with a clear vision of the system you want to create. The system's design is a crucial image in your head. If you're really luckier, the design's clarity makes it into the first release.
- Something goes wrong after that. As if it were a piece of poor meat, the programme begins to decay. The rot spreads and expands as time passes. In the code, ugly festering sores and boils accumulate, making it more difficult to maintain.
- Developers and front-line managers cry out for a redesign when the sheer effort necessary to make even the simplest of modifications becomes too much.
- Redesigns of this nature are rarely successful. Despite their best efforts, the designers discover that they are aiming at a moving target. The old system is still evolving and changing; therefore the new design needs to stay up. The new design develops warts and ulcers before it is released for the first time.

2.1.2 Design Smells

- When the programme begins to emit any of the following aromas, you know it's rotting :

1. **Rigidity :** It's difficult to update the system because every change requires several other changes in other parts of the system.
2. **Fragility :** Changes cause the system to fail in locations where there is no conceptual link to the modified section.

3. **Immobility :** It's difficult to break the system down into components that can be reused in other systems.
4. **Viscosity :** It is more difficult to do things correctly than it is to do things incorrectly.
5. **Needless complexity :** Infrastructure that provides no direct advantage is included in the design.
6. **Needless repetition :** Repeating structures in the design which could be united under a single abstraction.
7. **Opacity :** It's difficult to read and understand. It's not clear what it's trying to say.

1. Rigidity

- Rigidity refers to software's tendency to be difficult to alter, even in little ways. If a single change produces a cascade of subsequent modifications in dependent modules, the design is stiff. The design becomes increasingly stiff as the number of modules that must be altered increases.
- In one way or another, most developers have encountered this problem. They are requested to make a seemingly insignificant alteration. They examine the transition and generate a credible estimate of the work that will be required. However, as they work through the transition, they discover that the adjustment has unanticipated consequences.
- They find themselves tracking the changeover vast swaths of code, changing considerably more modules than they anticipated. Ultimately, the modifications take significantly longer than anticipated. When questioned why their estimate was so low, they say, "It was a lot more complicated than I imagined!" as is typical of software developers.

2. Fragility

- When a single change is made to a programme, it has a potential to break in multiple places.
- Frequently, the new difficulties arise in regions that have no conceptual connection to the changing domain. Fixing those issues causes other issues and the development team starts to resemble a dog chasing its tail. As a module's fragility grows, the possibility of an unforeseen problem arising from a change approaches certainty.

3. Immobility

- When a design has pieces that could be beneficial in other systems but the work and risk of isolating those parts from the original system is too considerable, the design becomes immobile. This is a regrettable but all-too-common occurrence.

4. Viscosity

- There are two types of viscosity : Software viscosity and environment viscosity.
- When presented with a change, developers frequently come up with multiple options. Some methods keep the design, whereas others don't (i.e., they are hacks.) The viscosity of the design is high when design-preserving methods are more difficult to implement than hacks. It is simple to make a mistake, but difficult to make the correct decision. We want to make it easier to make updates to our programme that keeps the design intact.
- When the development environment is slow and inefficient, the environment becomes viscous. If build times are excessively long, for example, developers may be motivated to make changes that avoid huge recompiles, even if those changes do not respect the design.
- Developers will be incentivized to make modifications that need as few check-ins as possible if the source-code management system requires hours to check in just a few files, regardless of whether the design is kept.
- A viscous project is one in which the software design is difficult to maintain in either circumstance. We want to design technologies and project settings that make it simple to keep the design intact.

5. Needless complexity

- When a design has aspects that aren't currently functional, it adds unnecessary complexity. This typically occurs when software developers anticipate changes in needs and provide features in the software to handle those changes. At first, this may seem like a good thing to do.
- After all, anticipating future changes should keep our code adaptable and avoid nightmare adjustments down the road.
- Unfortunately, the result is frequently the polar opposite. Too many eventualities are planned for and the design gets cluttered with structures that are never used. Some of those preparations may pay off, but the majority of them will not. Meanwhile, the weight of these underutilized design features is carried by the design. As a result, the programme is complicated and difficult to comprehend.

6. Needless repetition

- Cut and paste may be handy for text editing, but they can be devastating for code editing. Software systems are frequently based on dozens or hundreds of code parts that are repeated.
- This is how it goes :
  - Ralph needs to develop some code that breaks the arvadent. He hunts for a suitable piece of code in other portions of the code where he suspects more arvadent fravling has occurred. He copies and pastes that code into his module, making the necessary changes.
  - Ralph had no idea that the code he scraped up with his mouse had been put there by Todd, who had scraped it out of a Lilly module. Lilly was the first to fravle an arvadent, but she quickly learned that it was comparable to fravling a garnatosh. She found some garnatosh-related code someplace, cut and pasted it into her module and tweaked it as needed.
  - When the same code occurs in slightly different versions over and over again, the developers are missing an abstraction. Finding and eliminating all of the repetition with an appropriate abstraction may not be high on their priority list, but it would go a long way toward making the system easier to understand and manage.
- When there is redundant code in the system, replacing it might be a difficult task. Every time a bug is discovered in such a repeating unit, it must be fixed. The fix isn't always the same because each iteration is slightly different from the last.

7. Opacity

- The tendency of a module to be difficult to understand is called opacity. Code can be written in a way that is clear and expressive, or it might be written in a way that is opaque and complex. Code that changes over time tends to get increasingly obfuscated as it becomes older. To reduce opacity to a minimal, a persistent effort to keep the code clear and expressive is essential.
- When developers initially start writing a module, the code may appear to be straightforward. That is because they have immersed themselves in it and have a deep understanding of it.
- They may return to that module later, after the closeness has worn off and wonder how they could have written anything so bad. To avoid this, developers must put themselves in the shoes of their readers and make a concerted effort to refactor their

code so that it is understandable to them. They should also have their code checked by others.

2.1.2.1 What Causes Software to Degrade ?

- Designs degrade in non-agile situations because requirements evolve in ways that the original design did not anticipate. These changes are frequently required rapidly, and they may be implemented by developers unfamiliar with the original design concept.
- So, while the design update is functional, it is in some ways incompatible with the original design. As the revisions progress, these infractions mount up, and the design begins to stink.
- However, we cannot blame the design's degeneration on the requirements drifting. As software engineers, we are fully aware that needs are constantly changing. The majority of us recognize that the requirements are the project's most volatile elements.
- It is our designs and procedures that are at fault if our designs fail owing to the constant deluge of shifting needs. We need to figure out a way to make our designs resistant to such alterations and use methods to keep them from deteriorating.

2.1.2.2 Agile Teams Don't Let Software Degrade

- Change is what an agile team thrives on. The team makes a little initial investment, so it isn't invested in an out-of-date design. Instead, they maintain the system's architecture as clean and basic as possible, and they back it up with a large number of unit and acceptance tests.
- This keeps the design adaptable and changeable. The team takes advantage of this flexibility to enhance the design over time, so that each iteration finishes with a system that is as suitable as possible for the needs in that iteration.

2.1.3 "Copy" Program

- The above arguments can be illustrated by watching design decay. Assume your boss approaches you early on a Monday morning and requests that you develop software that copies characters from the keyboard to the printer. After some fast mental calculations, you determine that this will require fewer than 10 lines of code.
- The time spent on design and coding should be far less than one hour. This programme should take you about a week to complete, with cross-functional group meetings, quality education sessions, daily group progress meetings, and the three current problems in the area. You must, however, multiply your estimates by three.

2.1.3.1 Initial Design

- You have some free time before the process review meeting, so you decide to sketch out a plan for the programme. You create the structure chart using structured design.

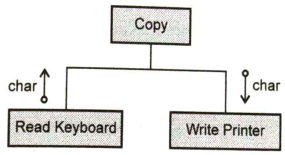

Fig. 2.1.1 Program structure chart for Copy

- The application is divided into three modules, or subprograms. The other two are called by the Copy module. Characters are retrieved from the Read Keyboard module and routed to the Write Printer module by the copy software. You examine your design and determine that it is satisfactory.
- You arrive a little early on Tuesday so that you can finish the Copy program. Unfortunately, one of the field crises has worsened overnight, and you must report to the lab to assist in the debugging of a problem. You manage to enter in the code for the Copy program during your lunch break, which you ultimately take at 3 p.m. The result is Listing 2 - 1.

Listing 2 - 1

The Copy Program

```
void Copy()
{
    int d;
    while ((d=RdKbd()) != EOF)
        WrtPrt(d);
}
```

- You have just saved the edit when you realize you're already late for an important meeting. You already know this is a big one; they'll be discussing the enormity of zero defects.
- Next day when you to office early, you open source code of Copy program and compile it. It compiles without problems the first time.

- Next day you test your Copy program after spending four hours on the phone with a service technician, taking him through the remote debugging and error logging instructions in one of the system's more obscure components. It's the first time it's worked ! It's also a good thing, because your new co-op student has just deleted the master source code directory from the server, and you'll have to go retrieve the most recent backup tapes and restore it. Naturally, the most recent complete backup was three months ago and you have ninety-four incremental backups to restore on top of that.
- The following day, loading of Copy program into source code control system is carried out. Naturally, the program is a huge success and is implemented throughout firm. Your reputation as a top programmer has been reaffirmed and you are ecstatic with your accomplishments.

2.1.3.2 Changing Requirements

- A few months later, your manager approaches you and asks that the Copy program should be able to read from the paper tape reader on occasion. Your teeth clench and your eyes roll. You may be confused as to why the requirements are always shifting. Your program was not created to be used with a paper tape reader! You advise your supervisor that such modifications will impact the elegance of your design. Your employer, on the other hand, is adamant. He claims that customers will really need to read characters from the paper tape reader.
- You want to make the Copy function take a boolean argument. If true, you would read from the paper tape reader; if false, you would continue to read from the keyboard. You can't modify the interface because there are so many other programs that use the Copy programme currently.
- Changing the interface would require recompiling and retesting for weeks. You would be lynched by the system test engineers alone, not to mention the seven people in the configuration control group. And the process cops would have a fun day enforcing all types of code reviews for every module named Copy !
- You will make use of the ? : operator, in order to let Copy program known to read from paper tape reader and which is the best and most helpful feature of the C programming language. The outcome is shown in Listing 2 - 2.

Listing 2 - 2

First modification of Copy program

```
bool ptFlag = false;
// remember to reset this flag
void Copy()

{
    int d;
    while ((d=(ptflag ? RdPt() : RdKbd())) != EOF)
    WrtPrt(d);
}
```

- To read from the paper tape reader, Copy callers must first set the ptFlag to true. They can then call Copy, which will gladly read from the paper tape reader.
- If the caller does not reset the ptFlag when Copy returns, the following caller may read from the paper tape reader instead of the keyboard. You've added an appropriate comment to remind the programmers of their responsibility to reset this flag.
- You deliver your program to critical acclaim once more. It's even more popular than before, and swarms of eager programmers are eagerly awaiting the chance to use it.
- After few day, customer changes in the requirements that sometimes clients requests the Copy programs to output on paper tape punch.
- The continuous changes in requirement make negative effect on elegance of design and which makes it difficult to maintain by the end of the year.
- But at the end you have to incorporate these changes into software. These changes are same as done before. Only another global and another ? : operator are required.
- The outcome of your efforts is shown in Listing 2 - 3.

Listing 2 - 3

```
bool ptFlag = false;
bool punchFlag = false;
// remember to reset these flags
void Copy()
{
    int d;
    while ((d=(ptflag ? RdPt() : RdKbd())) != EOF)
        punchFlag ? WrtPunch(d) : WrtPrt(d);
}
```

- The concern in this case that structure of the program is collapsing. In case input device changes again, you will have to rewrite the conditional while loop.

2.1.3.3 Expect Changes

- I will let you decide how much of what was said above was satirical exaggeration. The goal of the story was to demonstrate how a program's design may quickly degenerate in the face of change.
- The Copy program's original design was simple and attractive. After only two changes, it has started to show the signs of Rigidity, Fragility, Complexity, Immobility, Redundancy, and Opacity. This pattern will very definitely continue and the programme will devolve into a shambles.
- We could blame it on the changes if we wanted to. It is point of argument that the software was well-designed for the original spec, but that subsequent revisions to the spec caused the design to degrade. This, however, ignores one of the most important aspects of software development : requirements are constantly changing!
- Remember that the requirements are the most volatile aspect of most software projects. The requirements are in a constant state of change. This is a fact that we must embrace as developers ! We live in a world where requirements are always changing, and it is our job to ensure that our software can keep up. We are not agile if the design of our software degrades as a result of changing requirements.

2.1.3.4 Agile Design of the Copy Program

- With the code in Listing 2 - 1, an agile development might start in the same way. When the agile developers were asked to make the programme read from a paper tape reader, they would have changed the design to be more adaptable to such changes. The end product could have looked like Listing 2 - 4.

Listing 2 - 4

Agile version 2 of Copy

```
class Reader
{
    public:
        virtual int read() = 0;
};

class KeyboardReader : public Reader
```



```
{
    public:
        virtual int read() {return RdKbd();}
};

KeyboardReader GdefaultReader;

void Copy(Reader& reader = GdefaultReader)
{
        int d;
        while ((d=reader.read()) != EOF)
        WrtPrt(d);
}
```

- Rather than patching the design to meet the new need, the team takes use of the chance to improve the design so that it can withstand such changes in the future. When the boss requests a new type of input device in the future, the team will be able to reply in a way that does not degrade the copy program.
- The Open - Closed Principle (OCP) has been observed by the team. This idea urges us to design our modules in such a way that they can be extended without needing to be modified. That is precisely what the team has accomplished. Without altering the Copy program, any new input device requested by the boss can be delivered.
- It's worth noting, though, that when the team first developed the module, they didn't try to predict how the program would change. Instead, they wrote it in the most straightforward manner possible. It wasn't until the needs changed that they updated the module's design to make it more adaptable to such changes. They could be said to have only done half the job.
- They could have defended themselves from different output devices at the same time they were protecting themselves from diverse input devices. The team, on the other hand, has no idea whether the output devices will ever change. Adding the extra protection now would be time-consuming labour that would serve no current use. It is obvious that if such protection is required, it will be simple to implement. So there's no reason to include it right now.

2.1.3.5 How Did the Agile Developers Know What to Do ?

- In listing 2 - 4, the agile developers created an abstract class to shield themselves from changes to the input device. How did they figure out how to accomplish it ? One of the fundamental tenets of object-oriented design has anything to do with this.

- Because of the direction of its dependencies, the Copy program's initial design is inflexible. Look at Fig. 2.1.1 once more. It's worth noting that the Copy module is completely dependant on the *KeyboardReader* and *PrinterWriter*. This application's Copy module is a high-level module. It establishes the application's policies.
- It is capable of copying characters. Unfortunately, it has also been made dependant on the keyboard and printer's low-level information. As a result, when low-level specifics change, the high-level policy changes as well.
- Once the inflexibility was disclosed, the agile developers saw that the reliance from the Copy module to the input device needed to be inverted, allowing Copy to be independent of the input device. The STRATEGY4 pattern was then used to achieve the necessary inversion.
- So, in a nutshell, agile developers knew what they needed to do because
  1. They discovered the problem by following agile practices;
  2. They identified the problem by applying design principles; and
  3. They solved the problem by applying the appropriate design pattern.
- The act of design is the interaction between these three parts of software development.

2.1.4 Maintaining the Best Possible Design

- Agile developers are committed to maintaining a design that is both acceptable and tidy. This isn't a hasty or speculative decision. The design is not "cleaned up" every few weeks by agile developers. Every day, every hour, and even every minute, they maintain the software as clear, straightforward, and expressive as possible. They never allowed the degradation to start.
- The approach that agile developers have toward software design is similar to that of surgeons toward sterile procedures. Surgery is only feasible because of sterile procedures. The risk of infection would be simply too high to bear without it. The same can be said for agile developers when it comes to their designs.
- The danger of allowing even the least bit of rot to start is too great to bear. The design must be kept clean, and because the source code is the most visible manifestation of the design, it must also be kept clean. We can't accept code rot as software developers since we're professionals.

2.2 SRP : The Single - Responsibility Principle

- Consider the bowling game. For the majority of its development, the Game class

was responsible for two distinct tasks. It was computing the score and keeping track of the current frame. RCM and RSK eventually divided these two roles into two categories. The Game was given the task of keeping track of frames, while the Scorer was given the task of calculating the score.

- Why was it necessary to divide these two tasks into two distinct classes ? Because every obligation is a change axis. When the requirements change, it will be reflected in a shift in responsibilities among the courses. There will be multiple reasons for a class to alter if it takes on multiple responsibilities.
- When a class has many obligations, the responsibilities become linked. Changes in one obligation may damage or obstruct the class's capacity to meet the demands of the others. When this form of coupling occurs, designs become brittle and break in unanticipated ways.
- Take, for example, the design in Fig. 2.2.1. There are two methods in the Rectangle class. One creates a rectangle on the screen, while the other calculates the rectangle's area.

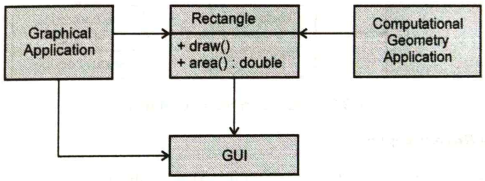

Fig. 2.2.1 More than one responsibility

- The Rectangle class is used in two different applications. Computational geometry is one of the applications. Rectangle is used to assist it with geometric shape mathematics. The rectangle is never drawn on the screen. The other programme is of a graphical kind. It might do some computational geometry as well, but it always draws the rectangle on the screen.
- The Single-Responsibility Principle is violated by this design (SRP). The Rectangle class is responsible for two things. The first job is to create a mathematical model of a rectangle's geometry. The rectangle must be rendered on a graphical user interface as the second obligation.
- This violation of the SRP results in a slew of issues. The GUI must first be included in the computational geometry application. If this were a C++ application, the GUI would have to be linked in, which would take time to link, compile and allocate

memory. The .class files for the GUI of a Java programme must be distributed to the target platform.

- Second, if a change to the *GraphicalApplication* causes the Rectangle to change for whatever reason, the *ComputationalGeometryApplication* may need to be rebuilt, retested, and redeployed. If we forget to do so, the application may malfunction in unexpected ways.
- Separating the two roles into two distinct classes, as shown in Fig. 2.2.2, is a preferable design. Rectangle's computational parts are moved to the GeometricRectangle class in this design. Changes to the rendering of rectangles no longer have an impact on the computational geometry application.

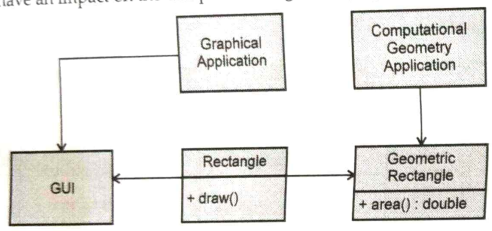

Fig. 2.2.2 Separated responsibilities

2.2.1 What is a Responsibility ?

- A responsibility is defined as "a justification for change" in the context of the SRP. If you can think of more than one reason to change classes, that class has multiple responsibilities.
- This can be difficult to notice at times. We're used to thinking of accountability in terms of groups. Consider the modem interface in Listing 2 - 5, for example. The vast majority of us will agree that this user interface appears to be totally fair. It declares four functions that are unmistakably modem functions.

Listing 2 - 5

Modem.java -- SRP Violation

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```


- However, two obligations are displayed here. The administration of connections is the first task. Data communication is the second. The dial and hang up operations control the modem's connection, while the send and receive capabilities send and receive data. Is it possible to separate these two responsibilities ? That is dependent on how the application evolves.
- If the application's signature changes in a way that affects the connection functions' signature, the design will smell like rigidity since the classes that call send and receive will have to be recompiled and redeployed more frequently than we'd like. In that instance, as indicated in Fig. 2.2.3, the two roles should be divided. This prevents the client applications from combining the two tasks.

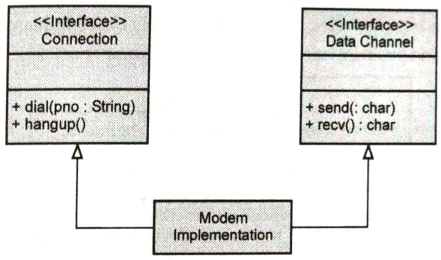

Fig. 2.2.3 Separated modem interface

- However, if the application does not changes in such a way that the two roles change at different times, there is no need to divide them. Separating them would definitely smell like needless complexity.
- There's a corollary to this. Only if the changes actually happen is an axis of change an axis of change. If there is no symptom, it is not a good idea to use the SRP or any other principle for that matter.

2.2.2 Separating Coupled Responsibilities

- In Fig. 2.2.3, you'll notice that both tasks are combined in the *ModemImplementation* class. This is not ideal, but it may be unavoidable. There are many causes, usually related to hardware or operating system characteristics that require us to couple things we don't want to pair.
- However, we have separated the notions in terms of the rest of the application by separating their interfaces. The *ModemImplementation* class may appear to be a kludge or a wart, but notice how all dependencies flow away from it.

- Nobody should be reliant on this group. It is unnecessary for anyone other than the main character to be aware of its existence. As a result, we have hidden the unsightly part behind a fence. Its ugliness doesn't have to spill over into the rest of the programme.

2.2.3 Persistence

- Fig. 2.2.4 depicts a common SRP violation. Business rules and persistence control are both contained in the Employee class. It's nearly never a good idea to blend these two tasks.
- Business rules change regularly, and while persistence does not change as frequently as business rules, it does so for entirely different reasons. It's a recipe for disaster to bind business rules to the persistence component.

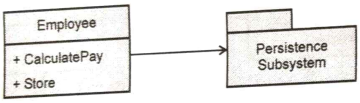

Fig. 2.2.4 Coupled persistence

- Fortunately, as we saw earlier, test-driven development forces these two roles to be separated long before the design starts to stink. However, if the tests did not compel the separation and the smells of Rigidity and Fragility became too strong, the design should be refactored to separate the two roles using the FACADE or PROXY patterns.

2.3 OCP : The Open - Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.
- The design smells of rigidity when a single modification to a programme causes a cascade of changes to dependent modules.
- The OCP recommends that we refactor the system such that future changes of this nature do not result in additional modifications. If the OCP is properly implemented, subsequent changes of this nature can be made by adding new code rather to altering existing code.
- This may appear to be the golden unattainable ideal of motherhood and apple pie, but there are several rather simple and efficient techniques for attaining that goal.

2.3.1 Description

- There are two main characteristics of modules that follow the Open - Closed Principle. They are :

1. Open for extension

- This signifies that the module's functionality can be expanded. As the application's requirements evolve, we can add new behaviors to the module to meet those requirements. In other words, we can change the module's behaviour.

2. Closed for modification

- Extending a module's behaviour does not need changes to the module's source or binary code. The module's binary executable version, whether in a linkable library, a DLL or a Java.jar, is unaffected.
- These two characteristics appear to be at odds with one another. Changing the source code of a module is the most common technique to increase its functionality.
- A module's behaviour is typically believed to be fixed if it can't be modified. How is it feasible to change the behaviour of a module without modifying the source code ? How can we change a module's behaviour without changing the module itself ?

2.3.2 Abstraction is the Key

- It is possible to design fixed abstractions in C++, Java, or any other OOPL3 that represent an unlimited set of possible behaviours. The abstractions are abstract base classes and all the derivative classes reflect the unlimited set of conceivable behaviours.

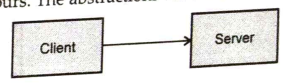

Fig. 2.3.1 Client is not open and closed

- A module has the ability to alter an abstraction. Because it is based on a fixed abstraction, such a module can be closed for modification. However, by establishing additional derivatives of the abstraction, the behaviour of that module can be extended.
- Fig. 2.3.1 shows a simple design that does not conform to the OCP. Both the Client and Server classes are concrete. The Client class uses the Server class. If we wish for a Client object to use a different server object, then the Client class must be changed to name the new server class.
- Fig. 2.3.2 depicts the relevant OCP-compliant design. ClientInterface is an abstract class having abstract member functions in this scenario. This abstraction is used by

the Client class; however, Client class objects will use objects from the derivative Server class.

- A new derivative of the ClientInterface class might be built if we wish Client objects to use a different server class. The Client class can be left alone.
- The Client has some tasks to complete, and it can express those tasks using the abstract interface provided by ClientInterface. ClientInterface subtypes can implement the interface in whatever way they want. As a result, by establishing new subtypes of ClientInterface, the functionality defined in Client can be extended and adjusted.

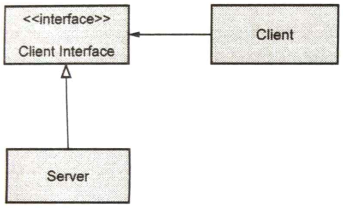

Fig. 2.3.2 Client is both open and closed - strategy pattern

- You might be wondering why *ClientInterface* was given that name. Why didn't I just call it AbstractServer ? The reason for this is that abstract classes are more closely related with their customers than with the classes that implement them, as we will see later.

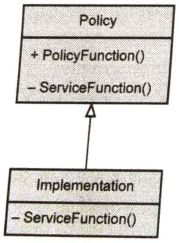

Fig. 2.3.3 Base class is open and closed - template method pattern

- A different construction is shown in Fig. 2.3.3. The Policy class has a set of concrete public functions that are used to implement a policy. Similar to the Client's functions in Fig. 2.3.2. Those policy functions, as before, specify some work to be done in terms of abstract interfaces.

- The abstract interfaces, on the other hand, are part of the Policy class. They'd be pure virtual functions in C++ and abstract methods in Java, respectively. These features are implemented in Policy subclasses. As a result, by developing additional derivatives of the Policy class, the behaviours provided within Policy can be extended or modified.
- The most popular ways to meet the OCP are these two patterns. They represent a clear distinction between generic functionality and its specialized implementation.

2.3.3 Shape Application

- Many books on OOD have used the following example. It's the well-known "Shape" example. It's most commonly used to demonstrate how polymorphism works. This time, however, we will utilize it to explain the OCP.
- We have an application that must be able to draw circles and squares on a standard GUI. The circles and squares must be drawn in a particular order. A list of the circles and squares will be created in the appropriate order, and the program must walk the list in that order and draw each circle or square.

2.3.3.1 Violating the OCP

- We have a program that needs to be able to draw circles and squares on a conventional graphical user interface. There is a specific order in which the circles and squares must be drawn. A list of the circles and squares will be constructed in the proper sequence, and the program will have to travel through the list and draw each circle or square in that order.
- We could handle this problem in C using procedural techniques that do not comply to the OCP, as indicated in Listing 2 - 6. We can see a group of data structures that all have the same first member but differ in other ways. The type code indicates the data structure as a circle or a square in the first element of each.

Listing 2 - 6

Procedural Solution to the Square/Circle Problem

```
--shape.h-----------------------------------
enum ShapeType {circle, square};

struct Shape
{
   ShapeType itsType;
};
```

```
--circle.h----------------------------------

struct Circle
{
    ShapeType itsType;
    double itsRadius;
    Point itsCenter;
};

void DrawCircle(struct Circle*);

--square.h----------------------------------

struct Square
{
    ShapeType itsType;
    double itsSide;
    Point itsTopLeft;
};

void DrawSquare(struct Square*);

--drawAllShapes.cc--------------------------
typedef struct Shape *ShapePointer;

void DrawAllShapes(ShapePointer list[], int n)
{
    int i;
    for (i=0; i<n; i++)
    {
        struct Shape* s = list[i];
        switch (s->itsType)
            {
            case square:
                DrawSquare((struct Square*)s);
            break;

            case circle:
                DrawCircle((struct Circle*)s);
            break;
            }
    }
}
```


- Because it cannot be closed against new types of shapes, the function DrawAllShapes does not comply with the OCP. If I wanted to extend this function to allow me to create a list of shapes that contained triangles, I'd have to change it. In fact, any new type of shape I needed to draw would necessitate modifying the code.
- This program is only an example. In actual life, the switch statement in the DrawAllShapes function would be repeated in many functions across the application, each one performing a slightly different task.
- There might be functions for dragging, stretching, shifting, and removing shapes, among other things. Adding a new shape to such an application entails searching for all switch statements (or if/else chains) and adding the new shape to each one.
- Furthermore, all switch statements and if/else chains are unlikely to be as well-structured as the one in DrawAllShapes. It's much more likely that the if statements' predicates will be coupled with logical operators, or that the switch statements' case clauses will be integrated to "simplify" local decision making.
- There may be functions that perform exactly the same thing to Squares as they do to Circles in some pathological cases. Switch/case statements and if/else chains would be absent from such functions. As a result, locating and comprehending all of the locations where the new shape must be introduced might be a difficult task.
- Consider the types of changes that might be required. To the ShapeType enum, we would have to add a new member. We would have to recompile all of the other shapes because they rely on the declaration of this enum. We would also have to recompile all the Shape-dependent modules.
- Not only must we modify the source code of any switch/case statements or if/else chains, but we must also change the binary files of all modules that use any of the Shape data structures (by recompilation). Any DLLs, shared libraries or other binary components must be redeployed if the binary files are changed.
- The mere act of adding a new form to the application triggers a chain reaction of changes to numerous source modules, as well as even more binary modules and binary components. Clearly, adding a new shape has a significant influence.

2.3.3.2 Bad Design

- Let's go through this again. Because the addition of Triangle causes Shape, Square, Circle, and DrawAllShapes to be recompiled and redeployed, the solution in Listing 2 - 6 is rigid. It's fragile because there will be a lot of other switch/case or

if/else statements that will be difficult to discover and comprehend.

- It's immobile because anyone trying to reuse DrawAllShapes in another application must bring Square and Circle with them, even if the new programme doesn't require them. As a result, Listing 2 - 6 has a lot of the negative design odours.

2.3.3.3 Conforming to the OCP

- Listing 2 - 7 provides the code for an OCP - compliant solution to the square/circle problem. In this scenario, we've created a Shape abstract class. Draw is the only abstract method in this abstract class. The Shape class has two derivatives : Circle and Square.

Listing 2 - 7

OOD solution to Square/Circle problem

```
class Shape
{
    public:
        virtual void Draw() const = 0;
};

class Square : public Shape
{
    public:
        virtual void Draw() const;
};

class Circle : public Shape
{
    public:
        virtual void Draw() const;
};

void DrawAllShapes(vector<Shape*>& list)
{
    vector<Shape*>::iterator i;
        for (i=list.begin(); i != list.end(); i++)
            (*i)->Draw();
}
```

- It's worth noting that adding a new derivative of the Form class to the DrawAllShapes function in Listing 2 - 7 will allow us to draw a new type of shape. There is no need to update the DrawAllShapes function. As a result, DrawAllShapes follows the OCP.

- Its functionality can be expanded without having to change it. Adding the Triangle class has no effect on any of the modules in this example. Clearly, some parts of the system must modify to accommodate the Triangle class, but the code presented here is unaffected.
- The Shape class would contain a lot more methods in a real application. Adding a new shape to the programme is still straightforward because all that is necessary is to build a new derivative and implement all of its features.
- There's no need to comb through the entire application looking for areas that need to be tweaked. This is not a fragile solution.
- The solution isn't either. There are no existing source modules that need to be changed, and there are no existing binary modules that need to be rebuilt, with one exception. Modifications must be made to the module that actually creates instances of the new shape derivative. This is usually handled by main, in a function called by main or in a method of an object generated by main.
- Finally, Immobile is not the answer. Any application can utilize DrawAllShapes without having to bring Square or Circle along for the ride. As a result, the solution possesses none of the previously listed characteristics of poor design.
- The OCP is followed in this software. Rather than modifying current code, it is altered by introducing new code. As a result, it does not go through the same chain of events as nonconforming programs. The only modifications necessary are the inclusion of a new module and a change to main that enables for the instantiation of new objects.

2.3.3.4 Natural Structure and Anticipation

- We could have constructed an abstraction to shield us from this kind of change if we had foreseen it. Listing 2 - 7's abstractions are more of a barrier than assistance in this type of development. This may come as a shock to you.
- What could be more natural than a Shape base class with derivatives Square and Circle ? What makes you think that natural model isn't the ideal one to use ? The reason is obvious : the model isn't natural in a system where form type is less important than ordering.
- This leads us to a disturbing conclusion. In general, no matter how "closed" a module is, there will always be some kind of change against which it is not closed. There is no model that is natural to all contexts ! Since closure cannot be complete, it must be strategic.

- As a result, we reach a dreadful conclusion. In general, no matter how "sealed" a module is, it will never be fully impenetrable to change. There is no such thing as a paradigm that works in every circumstance ! It must be strategic because there is no such thing as a perfect ending.
- This takes a certain amount of prescience derived from experience.
- The experienced designer hopes he knows the users and the industry well enough to judge the probability of different kinds of changes. He then invokes the OCP against the most probable changes.
- This necessitates a certain amount of experience-based foresight.
- The seasoned designer aims to have a good understanding of the users and the industry. Enough to assess the likelihood of certain types of changes after that, he The OCP is invoked in response to the most likely modifications.
- In addition, adhering to the OCP is costly. Creating adequate abstractions takes time and effort throughout development. These abstractions also add to the programme design's complexity. The degree of abstraction that the developers can afford is limited. Clearly, we want to limit the OCP's application to modifications that are likely to occur.
- How can we tell which changes are likely to occur ? We conduct the necessary research, ask the relevant questions, and apply our knowledge and common sense. After that, we wait for the adjustments to take place !

2.3.3.5 Safeguarding against Changes

- How do we safeguard ourselves against change ? We had a saying in the previous century. We'd "set the hooks" for any changes we anticipated may occur. We thought this would make our programme more adaptable.
- Our hooks, on the other hand, were frequently wrong. Worse, although not being used, they reeked of needless complexity that needed to be supported and maintained. This isn't a good situation. We don't want to clutter the design with too much abstraction. Rather, we frequently put off adding abstractions until we absolutely need them.

2.3.3.6 Stimulating Change

- If we choose to take the first bullet, it is in our best interests to get the bullets flying early and often. Before we get too far along the development route, we would like to know what sorts of adjustments are likely.

- The longer we wait to learn what types of changes are likely, the more difficult it will be to develop the necessary abstractions.
- As a result, we must encourage change.
  - We start by writing tests. Testing is one type of system application. We push the system to be testable by writing tests first. As a result, future changes in testability will not surprise us. We'll have created the abstractions that allow us to test the system. Many of these abstractions are likely to safeguard us from other types of changes in the future.
  - We develop in day-to-day cycles rather than weeks.
  - We build features before infrastructure and demonstrate them to stakeholders on a regular basis.
  - We develop the most important features first.
  - We deploy software frequently and early. We want to get it in front of our clients and users as soon as feasible.

2.3.3.7 Using Abstraction to Gain Explicit Closure

- So, we've already taken the first bullet. The user requests that all Circles be drawn before any Squares. We now wish to safeguard ourselves against similar shifts in the future.
- How can we protect the DrawAllShapes method from changes in drawing order ? Keep in mind that abstraction is the foundation of closure. As a result, we'll need some sort of "ordering abstraction" to make DrawAllShapes immune to ordering. This concept would serve as an abstract interface for expressing any potential ordering policy.
- An ordering policy states that it is possible to determine which of two things should be drawn first given any two objects. Shape has an abstract method called Precedes that we can define. This function accepts another Shape as a parameter and returns a bool value. If the Shape object that gets the message should be drawn before the Shape object supplied as an argument, the result is true.
- An overloaded operator function in C++ could be used to represent this function.
- With the ordering methods in place, Listing 2 - 8 demonstrates how the Shape class might look.
- We can sort and draw them in order now that we know how to establish the relative ordering of two Shape objects. The C++ code for this is shown in Listing 2 - 9.

Listing 2 - 8

Shape with ordering methods

```
class Shape
{
        public:
                virtual void Draw() const = 0;
                virtual bool Precedes(const Shape&) const = 0;

                bool operator<(const Shape& s) {return Precedes(s);
};
```

Listing 2 - 9

DrawAllShapes with Ordering

```
template <typename P>
class Lessp // utility for sorting containers of pointers.
{
        public:
                bool operator()(const P p, const P q) {return (*p) < (*q);}
};

void DrawAllShapes(vector<Shape*>& list)
{
        vector<Shape*> orderedList = list;

        sort(orderedList.begin(),
                orderedList.end(),
                Lessp<Shape*>());
        vector<Shape*>::const_iterator i;
        for (i=orderedList.begin(); i != orderedList.end(); i++)
                (*i)->Draw();
}
```

- This gives us a way to organize Shape objects and render them in the correct sequence. However, we still lack a good ordering abstraction. Individual Shape objects will have to override the Precedes function to determine ordering as it stands now. How would this function in practice ? In Circle, what kind of code would we write : Precedes in order to ensure that Circles are drawn first, followed by Squares ? Take a look at Listing 2 - 10.

Listing 2 - 10

Ordering a Circle

```
bool Circle::Precedes(const Shape& s) const
{
        if (dynamic_cast<Square*>(s))
                return true;
        else
                return false;
}
```

- It should be obvious that this function, like its siblings in the other Shape derivatives, does not follow the OCP. There's no way to protect them from new Shape derivatives. Every time a new Shape derivative is formed, all of the Precedes() routines must be updated. Of course, none of this matters if no new Shape derivatives are ever generated. On the other hand, if they're made frequently enough, this design will result in a lot of thrashing. We'd go with the first bullet once more.

2.3.3.8 Using a Data-Driven Approach to Achieve Closure

- We can utilize a table-driven technique to close the derivatives of Shape from each other's knowledge. One option is shown in Listing 2 - 11.

Listing 2 - 11

Table driven type ordering mechanism

```
#include <typeinfo>
#include <string>
#include <iostream>

using namespace std;

class Shape
{
        public:
                virtual void Draw() const = 0;
                bool Precedes(const Shape&) const;

                bool operator<(const Shape& s) const
                {return Precedes(s);}
        private:
                static const char* typeOrderTable[];
```

```
};

const char* Shape::typeOrderTable[] =
{
        typeid(Circle).name(),
        typeid(Square).name(),
        0
};
// This function searches a table for the class names.
// The table defines the order in which the
// shapes are to be drawn. Shapes that are not
// found always precede shapes that are found.
//
bool Shape::Precedes(const Shape& s) const
{
    const char* thisType = typeid(*this).name();
    const char* argType = typeid(s).name();
    bool done = false;
    int thisOrd = -1;
    int argOrd = -1;
    for (int i=0; !done; i++)
    {
        const char* tableEntry = typeOrderTable[i];
        if (tableEntry != 0)
        {
            if (strcmp(tableEntry, thisType) == 0)
                thisOrd = i;
            if (strcmp(tableEntry, argType) == 0)
                argOrd = i;
            if ((argOrd >= 0) && (thisOrd >= 0))
                done = true;
        }
        else // table entry == 0
            done = true;
    }
    return thisOrd < argOrd;
}
```

- We were able to successfully close the DrawAllShapes function against ordering difficulties in general, as well as each of the Shape derivatives against the generation of new Shape derivatives or a policy change that reorders the Shape objects by their type, by taking this approach.
- The table is the sole thing that is not closed against the order of the various Shapes. That table can be placed in its own module, distinct from the others, so that any

changes to it have no impact on the others. We can choose which table to utilize at link time in C++.

2.4 LSP : The Liskov Substitution Principle

- The LSP can be paraphrased as follows :

  SUBTYPES MUST BE SUBSTITUTABLE FOR THEIR BASE TYPES.
- Barbara Liskov first wrote this principle in 1988. She said, "What we're looking for is something along the lines of the substitution property : S is a subtype of T if there is an object o_2 of type T for each object o_1 of type S such that for any programs P described in terms of T, the behaviour of P is unaltered when o_1 is swapped for o_2".
- When you examine the repercussions of breaking this rule, you can see how important it is. Assume we have a function *f* that takes a pointer or reference to some base class B as an argument. Assume that there is some derivative D of B that, when provided to *f* as B, causes *f* to act badly. The LSP is then broken by D. In the presence of f, D is clearly Fragile.
- The writers off will be tempted to include a test for D so that when a D is handed to it, it behaves correctly. This test breaks the OCP since f is no longer closed to all of B's derivatives. These tests are a code smell caused by unskilled developers reacting to LSP violations.

2.4.1 Simple Example of a Violation of the LSP

- When the LSP is broken, Run-Time Type Information (RTTI) is frequently used in ways that contradict the OCP. An explicit if statement or an if/else chain is frequently used to determine the type of an object so that the appropriate behaviour can be chosen. Take a look at Listing 2 - 12.

Listing 2 - 12

A violation of LSP causing a violation of OCP

```
struct Point {double x,y;};

struct Shape
{
        enum ShapeType {square, circle} itsType;
        Shape(ShapeType t) : itsType(t) {}
};

struct Circle : public Shape
```

```
{
        Circle() : Shape(circle) {};
        void Draw() const;
        Point itsCenter;
        double itsRadius;
};
struct Square : public Shape

{
        Square() : Shape(square) {};
        void Draw() const;
        Point itsTopLeft;
        double itsSide;
};

void DrawShape(const Shape& s)
{
        if (s.itsType == Shape::square)
            static_cast<const Square&>(s).Draw();
        else if (s.itsType == Shape::circle)
            static_cast<const Circle&>(s).Draw();
}
```

- The OCP is clearly broken by the DrawShape method in Listing 2 - 12. It must be aware of every potential Shape derivative, and it must be updated anytime new Shape derivatives are formed. Many people, understandably, consider the structure of this function to be anathema to good design. What would motivate a coder to create such a function ?
- Take, for example, Joe the Engineer. Joe has examined object-oriented technology and has determined that the cost of polymorphism is too great to justify. As a result, he created the Shape class without any virtual functions. Square and Circle are structs that derive from Shape and contain Draw() functions, but they don't override any Shape functions.
- DrawShape must analyse its incoming Shape, establish its type, and then call the proper Draw function because Circle and Square are not substitutes for Shape.
- The fact that Square and Circle cannot be substituted for Shape is a violation of the LSP. DrawShape was obliged to violate the OCP as a result of this infraction. As a result, a breach of the LSP is a covert violation of the OCP.

2.4.2 Square and Rectangle, a More Subtle Violation

- There is, of course, other, far more subtle ways to break the LSP. Consider the application in Listing 2 - 10, which makes use of the Rectangle class.

Listing 2 - 13

Rectangle class

```
class Rectangle
{
        public:
                void SetWidth(double w) {itsWidth=w;}
                void SetHeight(double h) {itsHeight=w;}
                double GetHeight() const {return itsHeight;}
                double GetWidth() const {return itsWidth;}
        private:
                Point itsTopLeft;
                double itsWidth;
                double itsHeight;
};
```

- Assume that this application is well-liked and has been installed in a number of locations. Users want adjustments from time to time, as they do with all successful software. Users will one day expect the capacity to control squares as well as rectangles.
- Inheritance is often described as an IS-A relationship. To put it another way, if a new kind of object can be said to have an IS-A relationship with an old kind of object, then the new object's class should be derived from the old object's class.
- A square is a rectangle for all intents and purposes. As a result, the Square class is logically descended from the Rectangle class.

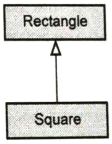

Fig. 2.4.1 Square inherits from rectangle

- One of the primary strategies of object-oriented analysis is the use of the IS-A relationship : Because a square is the same as a rectangle, the Square class should be derived from the Rectangle class.
- This kind of thinking, on the other hand, can lead to some minor but important issues. In most cases, we don't notice these issues until we see them in code.
- The fact that a Square does not require both itsHeight and itsWidth member

variables could be our first hint that something is awry. Nonetheless, Rectangle will pass them on to it. Clearly, this is a waste of resources.

- The majority of the time, such waste is minimal. However, if we had to create hundreds of thousands of Square objects, the amount of waste generated could be enormous.
- Let's consider that memory efficiency isn't a big deal to us. Other issues arise as a result of deriving Square from Rectangle. The SetWidth and SetHeight functions will be inherited by Square. Because the width and height of a square are the same, these functions are improper for it.
- This is a significant indicator that something is wrong. There is, however, a solution to avoid the issue.
- We could override SetWidth and SetHeight as follows :

```
void Square::SetWidth(double w)
{
        Rectangle::SetWidth(w);
        Rectangle::SetHeight(w);
}
void Square::SetHeight(double h)
{
        Rectangle::SetHeight(h);
        Rectangle::SetWidth(h);
}
```

- When the width of a Square object is changed, the height of the item changes as well. And as the height is changed, the width changes as well. As a result, the Square's invariants4 are preserved. The Square object will maintain its mathematical correctness.

```
Square s;
s.SetWidth(1); // Fortunately sets the height to 1 too.
s.SetHeight(2); // sets width and height to 2.
```

- But consider the following function :

```
void f(Rectangle& r)
{
        r.SetWidth(32); // calls Rectangle::SetWidth
}
```

- If we send a reference to a Square object into this function, it will get corrupted because the height will not be altered. This is an obvious breach of the LSP. For derivatives of its arguments, the f function does not work.

- The issue occurs because SetWidth and SetHeight in Rectangle were not declared virtual, and so are not polymorphic.
- This is a simple fix. When the construction of a derived class necessitates changes to the base class, however, it frequently indicates that the design is flawed. It unquestionably breaches the OCP.
- We may reply that the main design mistake was forgetting to make SetWidth and SetHeight virtual, which we are only now correcting. Setting the height and breadth of a rectangle are really primitive procedures, therefore this is difficult to justify. What rationale would we use to make them virtual if we weren't aware of Square's existence ?
- Still, let's assume we accept the argument and make the necessary changes to the classes. Listing 2 - 14 contains the final code.

Listing 2 - 14

Rectangle and Square that are Self-Consistent.

```
class Rectangle
{
    public:
        virtual void SetWidth(double w) {itsWidth=w;}
        virtual void SetHeight(double h) {itsHeight=h;}
        double GetHeight() const {return itsHeight;}
        double GetWidth() const {return itsWidth;}
    private:
        Point itsTopLeft
        double itsHeight;
        double itsWidth;
};

class Square : public Rectangle
{
    public:
        virtual void SetWidth(double w);
        virtual void SetHeight(double h);
};

void Square::SetWidth(double w)
{
    Rectangle::SetWidth(w);
    Rectangle::SetHeight(w);
}
```

```
void Square::SetHeight(double h)
{
    Rectangle::SetHeight(h);
    Rectangle::SetWidth(h);
}
```

2.4.2.1 Real Problem

- Square and Rectangle look to be working now. Whatever you do to a Square object, it will always maintain its mathematical square shape. And a mathematical rectangle will stay a mathematical rectangle no matter what you do to it. Furthermore, you can send a Square into a function that accepts a pointer or reference to a Rectangle and the Square will keep its shape and consistency.
- As a result, we can say that the design is now self-contained and correct. This conclusion, however, is incorrect. A self-consistent design isn't always consistent with all of its users ! Take a look at the following g function.

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.Area() == 20);
}
```

- This function calls the SetWidth and SetHeight members of a Rectangle it thinks it's got. When supplied a Rectangle, the method works perfectly, but when passed a Square, it reports an assertion error. So here's the issue: the author of g anticipated that modifying a Rectangle's width would not affect its height.
- It is obvious that increasing the width of a rectangle has no effect on its height! However, that assumption does not apply to all objects that can be provided as Rectangles. If you send an instance of a Square to a function like g, the function will fail since the author made that assumption. In terms of the Square/Rectangle hierarchy, function g is Fragile.
- Function g demonstrates that there are functions that accept pointers or references to Rectangle objects but are unable to work with Square objects. The relationship between Square and Rectangle violates the LSP since Square cannot be substituted for Rectangle for these functions.
- One could argue that the issue was with function g, and that the author had no right to assume that width and height were unrelated. The author of g, on the other hand, would disagree.

- The Rectangle is the argument to the g function. There are invariants, or truth assertions, that apply to a class named Rectangle, and one of them is that height and width are independent. This invariant was rightfully asserted by the author of g. Square's author is the one who has broken the invariant. Surprisingly, the inventor of Square did not break a Square invariant. The creator of Square broke an invariant of Rectangle by deriving Square from it !

2.4.2.2 Validity is not Intrinsic

- The LSP leads us to a crucial conclusion : a model cannot be properly verified if it is viewed in isolation. A model's validity can only be expressed in terms of its users. We discovered that the final versions of the Square and Rectangle classes were self - consistent and valid when we looked at them separately. The model, however, fell apart when we looked at them from the perspective of a coder who made acceptable assumptions about the base class.
- One cannot merely look at a solution in isolation while determining whether it is appropriate or not. It has to be viewed in light of the legitimate assumptions made by the design's users.
- Who can predict what legitimate assumptions a design's users will make ? The majority of these assumptions are difficult to predict. Indeed, if we tried to anticipate all of them, we'd probably end up with a system that smelled like Needless Complexity.
- As a result, it is frequently appropriate to defer all but the most blatant LSP violations until the relevant Fragility has been smelled, as it is with all other principles.

2.4.2.3 ISA is about Behavior

- So, what went wrong ? What went wrong with the Square and Rectangle's seemingly logical model ? After all, isn't a Rectangle a Square ? Isn't the IS-A relationship still valid ?
- Not in the opinion of the author of g ! A square may be a rectangle, but a Square object is not a Rectangle object from the perspective of g.
- Why ? Because the behaviour of a Square object differs from what g expects from a Rectangle object. A Square is not a Rectangle in terms of behaviour, and behaviour is what software is all about. The IS-A relationship in OOD, according to the LSP, refers to behaviour that can be fairly inferred and on which clients rely.

2.4.2.4 Design by Contract

- Many developers may be uneasy with the concept of "reasonably assumed" behaviour. How do you know what to expect from your customers ? There is a method for explicitly stating such reasonable assumptions and thereby enforcing the LSP. Bertrand Meyer popularised the concept, which he calls Design by Contract (DBC).
- The author of a class clearly states the contract for that class when using DBC. The contract tells the author of any client code about the expected behaviour. Each method's preconditions and postconditions are declared as part of the contract. In order for the procedure to run, the preconditions must be met.
- When the method is finished, the postconditions are guaranteed to be true.
- We can view the postcondition of Rectangle::SetWidth(double w) as follow :

```
assert((itsWidth == w) && (itsHeight == old.itsHeight));
```

- In this case, old refers to the value of the Rectangle prior to the call to SetWidth. According to Meyer, the rule for preconditions and postconditions of derivatives is now :

  Only one equal or weaker precondition and one equal or greater postcondition may be substituted in a routine redeclaration.
- In other words, while interacting with an object through its base-class interface, the user is solely aware of the base class's preconditions and postconditions. As a result, derived objects should not expect users to comply with preconditions that are more stringent than those imposed by the base class.
- That is to say, they must accept whatever the base class accepts. In addition, derived classes must meet all of the base class's postconditions. That is, neither their actions nor their outputs must break any of the underlying class's requirements. The output of the derived class must not confuse users of the base class.
- Because it does not enforce the constraint (itsHeight == old.itsHeight), the postcondition of Square::SetWidth(double w) is clearly weaker than the postcondition of Rectangle::SetWidth(double w). As a result, Square's SetWidth function breaks the underlying class's contract.
- Preconditions and postconditions are directly supported in some languages, such as Eiffel. You can declare them and have them verified by the runtime system. This is not a feature of C++ or Java. In these languages, we must manually check each method's preconditions and postconditions to ensure Meyer's rule is not broken.

Furthermore, documenting these preconditions and postconditions in the comments for each method can be quite useful.

2.4.2.5 Specifying Contracts in Unit Tests

- Unit tests can also be used to specify contracts. The unit tests exhaustively test the behaviour of a class, making the behaviour of the class obvious. Client code authors will want to go over the unit tests to see what they can properly expect about the classes they're utilizing.

2.4.3 Real Example

- Enough with the rectangles and squares! Is the LSP relevant to real-world software ? Let's have a look at a case study from a project on which I worked a few years back.

2.4.3.1 Motivation

- I bought a third-party class library with container classes in the early 1990s. The containers were generally related to Smalltalk's Bags and Sets. There were two different types of Set and two different types of Bag. The initial type, known as "bounded," was based on an array. The second was built on a linked list and was named "unbounded."
- The maximum amount of elements a BoundedSet could hold was determined in the function Object() { [native code] }. Within the BoundedSet, space for these elements was preallocated as an array. As a result, if the BoundedSet was successfully created, we knew it had adequate memory. It was extremely quick because it was based on an array.
- During regular operation, no memory allocations were made. We could also be guaranteed that operating the BoundedSet would not exhaust the heap because the memory had been preallocated. On the other hand, it was a waste of memory because it rarely used all of the space that it had preallocated.
- UnboundedSet, on the other hand, had no explicit limit on how many elements it could hold. The UnboundedSet would continue to accept elements as long as heap memory was available.
- As a result, it was extremely adaptable. It was also cost-effective because it only used the memory required to keep the elements it had at the time. It was also slow since it needed to allocate and deallocate memory on a regular basis. Finally, there was a risk that the heap would be depleted as a result of routine operations.

- The interfaces of these third-party classes irritated me. I didn't want my application code to be reliant on them because I planned to replace them with better classes in the future. As a result, as illustrated in Fig. 2.4.2, I wrapped the third-party containers in my own abstract interface.

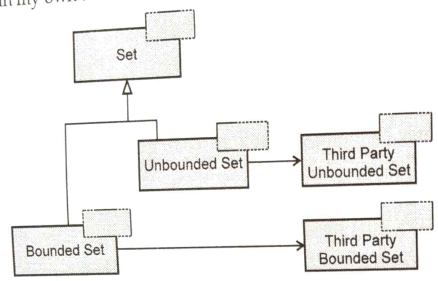

Fig. 2.4.2 Container class adapter layer

- As illustrated in Listing 2 - 15, I constructed an abstract class called Set that included pure virtual Add, Delete and IsMember operations. The unbounded and bounded variants of the two third-party sets were united by this structure, which allowed them to be accessed through a common interface.
- As a result, certain clients may take a Set<T>& parameter regardless of whether the actual Set it worked on was bounded or unbounded.

Listing 2 - 15

Abstract Set Class

```
template <class T>
class Set
{
    public:
        virtual void Add(const T&) = 0;
        virtual void Delete(const T&) = 0;
        virtual bool IsMember(const T&) const = 0;
};
```

A Real Example

Listing 2 - 16

PrintSet

```
template <class T>
void PrintSet(const Set<T>& s)
```



```
{
    for (Iterator<T>i(s); i; i++
        cout << (*i) << endl;
}
```

- It is a significant benefit to not have to know or care about the type of Set you are using. It means that the programmer can choose whatever type of Set is required in each case, and none of the client functions will be impacted by this selection.
- When memory is limited and performance is not crucial, the programmer should use an UnboundedSet; when memory is sufficient and speed is critical, the programmer should use a BoundedSet. Client methods will manipulate these objects using the interface of the base class Set and will not be aware of or care about the type of Set they are using.

2.4.3.2 Problem

- To this hierarchy, I wanted to add a PersistentSet. A persistent set is one that can be written out to a stream and then read back in by another application later.
- Unfortunately, the only third-party container with persistence that I had access to was not a template class. Objects generated from the abstract base class PersistentObject were instead accepted. Fig. 2.4.3 shows the hierarchy I developed.

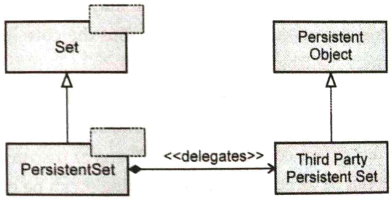

Fig. 2.4.3 PersistentSet hierarchy

- It's worth noting that PersistentSet contains a third-party persistent set instance to which it delegated all of its functions. As a result, calling Add on the PersistentSet merely delegated to the relevant method of the contained third-party persistent set.
- On the surface, everything appears to be in order. There is, nevertheless, an unfavourable implication. PersistentObject must be used to create new elements for the third-party persistent collection. Because PersistentSet simply delegated to a third-party persistent set, any member added to PersistentSet must be a descendant of PersistentObject. Set, on the other hand, has no such restriction in its interface.
- When a client adds members to the base class Set, the client has no way of knowing

if the Set is a PersistentSet or not. As a result, the client has no means of knowing if the elements it adds should be derived from PersistentObject or not.

- Consider the code for PersistentSet::Add() in Listing 2 - 17.

Listing 2 - 17

```
template <typename T>
void PersistentSet::Add(const T& t)
{
    PersistentObject& p =
        dynamic_cast<PersistentObject&>(t);
    itsThirdPartyPersistentSet.Add(p);
}
```

- This code makes it apparent that if a client attempts to add an object to my PersistentSet that is not derived from the class PersistentObject, a runtime error will occur. Bad cast will be thrown by dynamic cast. Exceptions are not expected to be thrown on Add by any of the existing clients of the abstract base class Set.
- This change to the hierarchy violates the LSP because a derivative of Set will confuse these functions.
- Is this an issue ? Certainly. When handed a PersistentSet, functions that previously never failed when passed a derivative of Set may now trigger runtime issues. Because the runtime issue occurs so far away from the core logic flaw, debugging this type of problem is challenging.
- The decision to send a PersistentSet into a method, or the decision to add an object to the PersistentSet that is not derived from PersistentObject, is the logic fault. In any situation, the real choice could be made millions of instructions before the Add method is called. It can be difficult to locate it. Fixing it can be even more difficult.

2.4.3.3 Solution that does not Conform to the LSP

- What are our options for resolving this issue ? I solved it by convention a few years back. That is to say, I did not solve the problem in source code. Rather, I defined a rule that PersistentSet and PersistentObject were unknown to the entire application.
- Only one module was aware of their existence. This module was in charge of reading and writing to the persistent storage all of the containers. When the contents of a container were to be written, they were copied into appropriate PersistentObject derivatives and then added to PersistentSets, which were subsequently saved on a stream.

- The process was inverted when a container required to be read from a stream. The PersistentObjects in the PersistentSet were removed and copied into ordinary (nonpersistent) objects, which were then added to a standard Set. This method may appear unnecessarily restrictive, but it was the only way I could think of to keep PersistentSet objects out of the interfaces of functions that wanted to add nonpersistent objects to them. Furthermore, it broke the remainder of the application's reliance on the concept of persistence.
- Is this a viable solution ? Not at all. Developers who did not comprehend the importance of the convention broke it in various places of the application. The problem with conventions is that they have to be resold to each developer on a regular basis.
- The convention will be broken if the developer has not learned the convention or does not agree with it. And a single infraction might jeopardies the entire system.

2.4.3.4 LSP-Compliant Solution

- What would I do now if I had to address this problem ? PersistentSet does not have an IS-A relationship with Set, and thus is not a legitimate derivation of Set, as I accept. As a result, I would partially split the hierarchies. There are several similarities between Set and PersistentSet. In reality, the Add technique is the only one that presents problems with LSP.
- What would I do now if I had to address this problem ? PersistentSet does not have an IS-A relationship with Set, and thus is not a legitimate derivation of Set, as I accept. As a result, I would partially split the hierarchies. There are several similarities between Set and PersistentSet. In reality, the Add technique is the only one that presents problems with LSP.

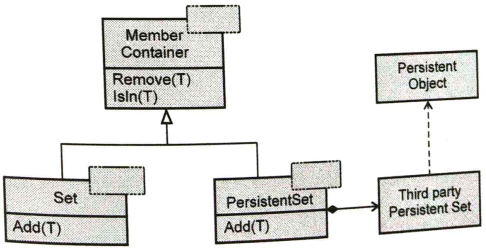

Fig. 2.4.4 LSP compliant solution

2.4.4 Factoring Instead of Deriving

- The Line and the LineSegment are another intriguing and perplexing case of inheritance. Take a look at Listings 2 - 18 and 2 - 19. These two classes appear to be natural candidates for public inheritance at first glance.
- Every member variable and function declared in Line is required by LineSegment. LineSegment also adds its own member function, GetLength, which alters the meaning of the IsOn function.
- Yet these two classes violate the LSP in a subtle way.

Listing 2 - 18

geometry/line.h

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/point.h"

class Line
{
    public:
        Line(const Point& p1, const Point& p2);

        double GetSlope() const;
        double GetIntercept() const; // Y Intercept
        Point GetP1() const {return itsP1;};
        Point GetP2() const {return itsP2;};
        virtual bool IsOn(const Point&) const;

    private:
        Point itsP1;
        Point itsP2;
};
#endif
```

Listing 2 - 19

geometry/lineseg.h

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
class LineSegment : public Line
{
    public:
        LineSegment(const Point& p1, const Point& p2);
        double GetLength() const;
```



```
        virtual bool IsOn(const Point&) const;
};
#endif
```

- A user of Line has the right to anticipate that it contains all colinear points. The point where the line crosses the y-axis, for example, is returned by the Intercept function. Users of Line have a right to assume that IsOn(Intercept()) == true because this point is parallel to the line. There are numerous instances of
- However, because of LineSegment, this sentence will fail.
- Why is this a significant issue ? Why not just deduce LineSegment from Line and deal with the nuances later ? This is a decision that must be made on a case-by-case basis. There are times when accepting a minor defect in polymorphic behaviour is preferable to attempting to bend the design into total LSP compliance.
- It's an engineering trade-off to accept compromise rather than strive for perfection. When compromise is more profitable than perfection, a competent engineer recognizes that. Conformance to the LSP, on the other hand, should not be taken lightly.
- A powerful technique to control complexity is to guarantee that a subclass will always work where its base classes are utilized. We must consider each subclass individually once it has been abandoned.
- There is a simple solution for the Line and LineSegment that demonstrates an important OOD tool. We can factor the common elements of both the Line and LineSegment classes into an abstract base class if we have access to both. Line and LineSegment are factored into the base class LinearObject in Listings 2 - 20 through 2 - 22.

Listing 2 - 20

geometry/linearobj.h

```
#ifndef GEOMETRY_LINEAR_OBJECT_H
#define GEOMETRY_LINEAR_OBJECT_H

#include "geometry/point.h"

class LinearObject
{
    public:
        LinearObject(const Point& p1, const Point& p2);

        double GetSlope() const;
```

```
        double GetIntercept() const;

        Point GetP1() const {return itsP1;};
        Point GetP2() const {return itsP2;};
        virtual int IsOn(const Point&) const = 0; // abstract.

    private:
        Point itsP1;
        Point itsP2;
};
#endif
```

Listing 2 - 21

geometry/line.h

```
#ifndef GEOMETRY_LINE_H
#define GEOMETRY_LINE_H
#include "geometry/linearobj.h"

class Line : public LinearObject
{
    public:
        Line(const Point& p1, const Point& p2);
        virtual bool IsOn(const Point&) const;
};
#endif
```

Listing 2 - 22

geometry/lineseg.h

```
#ifndef GEOMETRY_LINESEGMENT_H
#define GEOMETRY_LINESEGMENT_H
#include "geometry/linearobj.h"

class LineSegment : public LinearObject
{
    public:
        LineSegment(const Point& p1, const Point& p2);

        double GetLength() const;
        virtual bool IsOn(const Point&) const;
};
#endif
```


- Both Line and LineSegment are represented by LinearObject. With the exception of the IsOn method, which is pure virtual, it supplies the majority of the functionality and data members for both subclasses.
- LinearObject users are not permitted to assume that they are aware of the scope of the object they are using. As a result, they can easily accept either a Line or a LineSegment. Additionally, Line users will never have to deal with a LineSegment.
- Factoring is a design technique that works well when there isn't much code to write. We would not have had an easy time factoring out the LinearObject class if there were dozens of customers of the Line type presented in Listing 2 - 18. However, when factoring is possible, it is a powerful tool.
- If two subclasses' qualities can be factored out, there's a good chance that other classes will require those traits in the future. Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener say this about factoring :
- We can say that if a group of classes all support the same responsibility that duty should be passed down from a common superclass. Create a common superclass if one doesn't already exist, and assign the common responsibilities to it.
- After all, such a class is plainly useful - you've previously demonstrated that some classes will inherit the obligations. Isn't it possible that a future upgrade to your system will include a new subclass that supports those same tasks in a different way ? This new superclass will most likely be abstract. The characteristics of LinearObject can be used by an unexpected class, Ray, as seen in Listing 2 - 23. A Ray can be used in place of a LinearObject, and no LinearObject user will have any problems with it.

Listing 2 - 23

geometry/ray.h

```
#ifndef GEOMETRY_RAY_H
#define GEOMETRY_RAY_H

class Ray : public LinearObject
{
    public:
        Ray(const Point& p1, const Point& p2);
        virtual bool IsOn(const Point&) const;
};
#endif
```

2.4.5 Heuristics and Conventions

- There are a few simple heuristics that can help you figure out if there are any LSP violations. They're all about derived classes that take away functionality from their base classes in some way. A derivative that performs poorly in comparison to its parent. base is rarely interchangeable with that base, and hence violates the LSP.

2.4.5.1 Degenerate Functions in Derivatives

- Take a look at Listings 2 - 24. In Base, the f function is implemented. It is, however, degenerate in Derived. Presumably, the creator of Derived concluded that function f in a Derived was useless. Unfortunately, Base users are unaware that they should not call f, resulting in a substitution error.

Listing 2 - 24

A degenerate function in a derivative

```
public class Base
{
        public void f() {/*some code*/}
}

public class Derived extends Base
{
        public void f() {}
}
```

- Degenerate functions in derivatives aren't necessarily suggestive of an LSP violation, but they're worth investigating when they appear.

2.4.5.2 Throwing Exceptions from Derivatives

- Adding exceptions to methods of derived classes whose bases don't throw them is another sort of violation. If the users of the base classes aren't expecting exceptions, then adding them to the derivative methods isn't a good idea. Either the users' expectations must be changed or the derived classes must not throw exceptions.

2.5 DIP : The Dependency - Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

- Many people have wondered why I refer to this principle as "inversion" over the years. Because more traditional software development methodologies, such as Structured Analysis and Design, build software structures in which high-level modules depend on low-level modules and policy depends on detail, this is the case.
- One of these methods' objectives is to define the subprogram hierarchy, which explains how high-level modules call low-level modules.
- In comparison to the dependence structure that generally comes from traditional procedural approaches, the dependency structure of a well-designed, object-oriented application is "inverted."
- One of these methods' objectives is to define the subprogram hierarchy, which explains how high-level modules call low-level modules. However, because these modules rely on lower-level modules, changes to the lower-level modules might have a direct impact on the higher-level modules, forcing them to alter as well.
- This situation is ludicrous ! The high-level, policy-making modules should have an impact on the low-level, detailed modules. The modules that provide the high-level business rules should take precedence over the modules that contain the implementation details and should be independent of them. High-level modules should not, under any circumstances, rely on low-level modules.
- Furthermore, we want to be able to reuse high-level policy-setting modules. We have gotten really proficient at reusing low-level modules as subroutine libraries. When high-level modules rely on low-level modules, reusing those high-level modules in other contexts becomes extremely challenging.
- When the high-level modules are independent of the low-level modules, however, the high-level modules can easily be reused.
- This is a fundamental principle of framework design.

2.5.1 Layering

- All well-structured object-oriented architectures have clearly defined layers, with each layer delivering a coherent set of services through a well-defined and regulated interface.
- A designer might create a structure that looks like Fig. 2.5.1 based on a naive understanding of this statement. The high-level Policy layer employs a lower-level Mechanism layer, which uses a detailed-level Utility layer in this diagram. While this appears to be suitable, the Policy layer is vulnerable to changes all the way

down in the Utility layer, which is an insidious feature.

- Dependency is a two-way street. The Policy layer is transitively dependent on the Utility layer since it is dependent on something that is dependent on the Utility layer. This is a terrible situation.

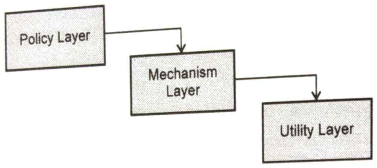

Fig. 2.5.1 Naive layering scheme

- A more acceptable model is shown in Fig. 2.5.2. For the services it requires, each of the upper-level layers specifies an abstract interface. These abstract interfaces are then used to create the lower-level layers. The abstract interface is used by each higher-level class to access the next-lowest layer. As a result, the upper layers are independent of the lower layers.
- Instead, the lower levels rely on the upper layers' abstract service interfaces. Not only is PolicyLayer's transitive dependent on UtilityLayer flawed, but so is PolicyLayer's direct dependency on MechanismLayer.

2.5.1.1 Inversion of Ownership

- It's worth noting that the inversion isn't only about dependencies; it's also about interface ownership. Utility libraries are frequently thought to have their own interfaces. However, when the DIP is used, we discover that clients tend to possess abstract interfaces, which their servers derive from.
- "Don't call us, we'll call you," is what the Hollywood principle is known as. The lower-level modules implement interfaces that are specified within the upper-level modules and invoked by them.
- PolicyLayer is unaffected by changes to MechanismLayer or UtilityLayer because of this ownership inversion. Furthermore, PolicyLayer can be employed in any context that defines PolicyServiceInterface-compliant lower-level modules. As a result of reversing the dependencies, we have produced a structure that is more adaptable, robust, and transportable at the same time.

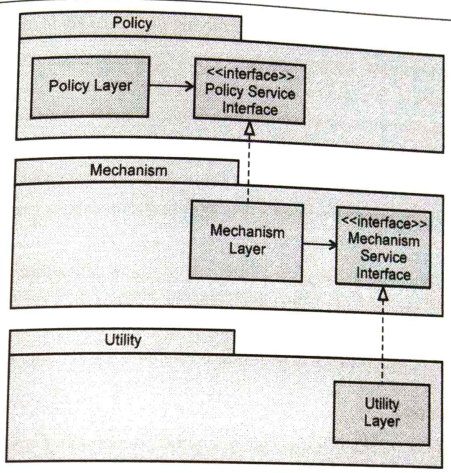

Fig. 2.5.2 Inverted layers

2.5.1.2 Depend on Abstractions

- The basic heuristic : "Depend on abstractions" is a slightly more simplistic, but still highly powerful, interpretation of the DIP. Simply said, this heuristic suggests that you should not rely on a concrete class and that all programme relationships should finish with an abstract class or an interface.
- According to this heuristic,
  - A pointer or reference to a concrete class should never be stored in a variable.
  - There should be no classes that inherit from concrete classes.
  - No method shall override a method that is implemented in one of its base classes.
- This heuristic is almost always broken at least once in every program. Someone has to construct the concrete class instances, and whichever module does so will be dependent on them. 3 Furthermore, there appears to be no reason to use this heuristic for nonvolatile but concrete classes. If a concrete class isn't going to change much and no other derivatives are going to be formed, then relying on it isn't going to do any harm.
- In most systems, the class that describes a string, for example, is concrete. It's the concrete class String in Java, for example. This is a non-volatile class. That is to say,

it does not alter frequently. As a result, relying on it directly is not harmful.

- Most concrete classes we write as part of an application programme, on the other hand, are volatile. We don't want to be reliant on those concrete classes directly. By putting them behind an abstract interface, you can isolate their volatility.
- This isn't a full-fledged answer. There are situations when a volatile class's interface must be modified, and this modification must be propagated to the abstract interface that represents the class. Such modifications shatter the abstract interface's isolation.
- Because of this, the heuristic is a little naive. If, on the other hand, we take the longer view and assume that client classes declare the service interfaces they require, the interface will only change when the client requires it. The client will be unaffected by changes to the classes that implement the abstract interface.

2.5.2 Simple Example

- Wherever one class sends a message to another, dependency inversion can be used. Consider the cases of the Button and the Lamp objects, for example.
- The Button item is aware of its surroundings. It evaluates if a user has "pressed" the Poll message after getting it. It makes no difference what kind of sensing mechanism is used. A button icon on a GUI, a physical button pressed by a human finger or even a motion detector in a home security system might all be examples.
- The Button object recognizes when it has been activated or deactivated by a user.
- The Lamp object affects the external environment. On receiving a TurnOn message, it illuminates a light of some kind. On receiving a TurnOff message, it extinguishes that light. The physical mechanism is unimportant. It could be an LED on a computer console, a mercury vapor lamp in a parking lot, or even the laser in a laser printer.
- How can we construct a system in which the Lamp object is controlled by the Button object ? A basic design is shown in Fig. 2.5.3. The Button object receives Poll messages, assesses whether the button has been pressed, and then sends the Lamp the TurnOn or TurnOff message.

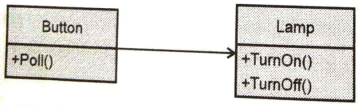

Fig. 2.5.3 Basic model of button and lamp

- What makes this so naïve ? Consider the Java code that this paradigm implies (Listing 2 - 25). It's worth noting that the Button class is completely reliant on the Lamp class. Changes to Lamp will have an impact on Button because of this reliance. Furthermore, you won't be able to utilize Button to operate a motor item. Button objects, and only Button objects, govern Lamp objects in this design.

Listing 2 - 25

Button.java

```
public class Button
{
    private Lamp itsLamp;
    public void poll()
    {
        if (/*some condition*/)
            itsLamp.turnOn();
    }
}
```

- This solution is not compliant with the DIP. The application's high-level policy has not been isolated from its low-level implementation. The abstractions and the details have not been separated. Without this separation, the high-level policies are forced to rely on the low-level modules, and the abstractions are forced to rely on the details.

2.5.2.1 Finding the Underlying Abstraction

- What is the policy at the highest level ? It is the abstraction that underpins the application, the truths that remain constant regardless of the details. It's the metaphor for the system within the system.
- The fundamental abstraction in the Button/Lamp example is to detect a user's on/off gesture and transfer that gesture to a target object. What method is used to recognize the user's gesture ? Irrelevant ! What is the object that you're aiming for ? Irrelevant ! These are minor facts that have no bearing on the abstraction.
- Inverting the dependent on the Lamp object can enhance the architecture in Fig. 2.5.3. The Button now has a link with something called a ButtonServer, as seen in Fig. 2.5.4. ButtonServer provides abstract ways for turning anything on or off that Button can use. The ButtonServer interface is implemented by Lamp.
- Lamp is now the one who is being depended on, rather than the one who is being depended on.

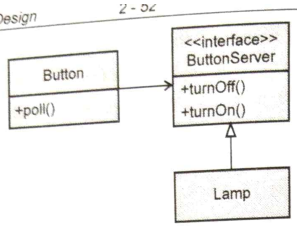

Fig. 2.5.4 Dependency inversion applied to lamp

- Fig. 2.5.4 illustrates how a Button can control any device that implements the ButtonServer interface. This provides us with a lot of options. It also means that Button objects will be able to control yet-to-be-created items.
- However, any object that has to be controlled by a Button is constrained by this technique. The ButtonServer interface must be implemented by such an object. This is problematic because these things could prefer to be controlled by a Switch or another item than a Button.
- We have made Lamp rely on a new detail-Button-by inverting the direction of the dependency and making the Lamp perform the depending instead of being depended on. Have we, or have we not ?
- ButtonServer is dependent on Lamp, while Lamp is not dependent on ButtonServer. A Lamp can be controlled by any object that understands how to use the ButtonServer interface. As a result, the reliance exists just in name. We can solve this by renaming ButtonServer to something more generic, such as SwitchableDevice.
- We can also keep Button and SwitchableDevice in different libraries, ensuring that using SwitchableDevice does not necessitate using Button.
- Nobody owns the interface in this scenario. We have a unique circumstance in which the interface can be utilised by a variety of clients and implemented by a variety of servers.
- As a result, the interface must be able to function independently of either group. We'd put it in a different namespace and library in C++. In Java we would put it in a separate package.

2.5.3 Furnace Example

- Let's take a look at a more fascinating scenario. Consider the software that might be used to regulate a furnace's regulator. By sending commands to a different IO channel, the programme can read the current temperature from one IO channel and

direct the furnace to switch on or off. The algorithm's structure might resemble that of Listing 2 - 26.

Listing 2 - 26

Simple algorithm for a thermostat

```
#define TERMOMETER 0x86
#define FURNACE 0x87
#define ENGAGE 1
#define DISENGAGE 0

void Regulate(double minTemp, double maxTemp)
{
    for(;;)
    {
        while (in(THERMOMETER) > minTemp)
            wait(1);
        out(FURNACE,ENGAGE);

        while (in(THERMOMETER) < maxTemp)
            wait(1);
        out(FURNACE,DISENGAGE);
    }
}
```

- The algorithm's high-level aim is clear, but the code is clogged with low-level details. This code could never be repurposed for use with other control hardware.
- Given the tiny size of the code, this may not be a significant loss. Even still, it's a pity that the algorithm is no longer available for reuse. Fig. 2.5.5 is what we would like to see if the dependencies were inverted.

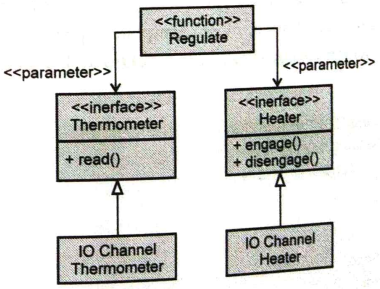

Fig. 2.5.5 Generic regulator

- This demonstrates that the regulate function accepts two arguments, both of which are interfaces. The Heater interface can be engaged and disengaged, and the Thermometer interface can be read. All the Regulate algorithm need is this. As illustrated in Listing 2 - 27, it can now be written.

Listing 2 - 27

Generic Regulator

```
void Regulate(Thermometer& t, Heater& h,
              double minTemp, double maxTemp)
{
    for(;;)
    {
        while (t.Read() > minTemp)
            wait(1);
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

- As a result, the dependencies have been inverted, and the high-level regulating policy is independent of any of the thermometer or furnace's precise features. The algorithm has a lot of reusability.

2.5.3.1 Dynamic Vs. Static Polymorphism

- Through the use of dynamic polymorphism, we were able to invert the dependencies and make Regulate general (i.e., abstract classes or interfaces). There is, however, another option. C++ templates provide a static version of polymorphism, which we could have used. Take a look at Listing 2 - 28.

Listing 2 - 28

```
template <typename THERMOMETER, typename HEATER>
class Regulate(THERMOMETER& t, HEATER& h,
               double minTemp, double maxTemp)
{
    for(;;)
    {
        while (t.Read() > minTemp)
            wait(1);
```



```
        h.Engage();

        while (t.Read() < maxTemp)
            wait(1);
        h.Disengage();
    }
}
```

- Without the expense of dynamic polymorphism, this provides the same inversion of dependencies. Read, Engage and Disengage are all nonvirtual methods in C++. In addition, any class that states that the template can utilize these methods. They don't have to share a common ancestor.
- Regulate does not rely on any specific implementation of these functions as a template. All that is required is that the class substituted for HEATER has an Engage and Disengage method, as well as a Read function in the class substituted for THERMOMETER.
- As a result, the classes must implement the template's interface. To put it another way, both Regulate and the classes it uses must agree on the same interface, and they both rely on it.
- Static polymorphism is a good way to break the source-code reliance, but it doesn't solve as many issues as dynamic polymorphism does.
- The template technique has the following drawbacks : (1) the types of HEATER and THERMOMETER cannot be modified at runtime; and (2) the use of a new type of HEATER or THERMOMETER will need recompilation and redeployment. Dynamic polymorphism should be preferable unless you have a very strict demand for speed.

2.6 ISP : The Interface - Segregation Principle

- This theory addresses the drawbacks of "fat" interfaces. Classes with "fat" interfaces are those with non-cohesive interfaces. In other words, the class's interfaces can be divided into groups of methods. Each group caters to a distinct set of customers. As a result, some clients use one set of member functions while others use the other.
- Although the ISP recognizes that some objects require non cohesive interfaces, it proposes that clients should not be aware of them as a single class. Clients should instead be aware of abstract base classes with consistent interfaces.

2.6.1 Interface Pollution

- Consider investing in a security system. There are Door objects in this system that can be locked and unlocked and that know whether they are open or closed.

Listing 2 - 29

Security Door

```
class Door
{
    public:
        virtual void Lock() = 0;
        virtual void Unlock() = 0;
        virtual bool IsDoorOpen() = 0;
};
```

- Clients can utilize objects that adhere to the Door interface without having to rely on specific implementations of Door because this class is abstract.
- Consider that one of these implementations, TimedDoor, needs to sound an alarm if the door is left open for an extended period of time. The TimedDoor object talks with another object called a Timer to do this.

Listing 2 - 30

```
class Timer
{
    public:
        void Register(int timeout, TimerClient* client);
};
class TimerClient
{
    public:
        virtual void TimeOut() = 0;
};
```

- When an object wants to be notified about a timeout, it uses the Timer's Register function. The time of the time-out and a pointer to a TimerClient object whose TimeOut method will be invoked when the time-out expires are the inputs to this function.
- How can we get the TimerClient class to interact with the TimedDoor class so that the TimedDoor's code is alerted when the timer expires ? There are various options available. A naive solution is shown in Fig. 2.6.1. Door, and hence TimedDoor, are forced to derive from TimerClient. TimerClient will be able to register with the Timer and get the TimeOut message as a result of this.

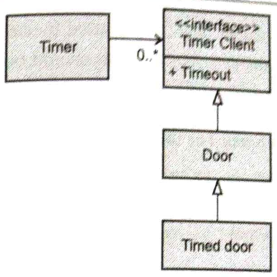

Fig. 2.6.1 Timer client at top of hierarchy

- Although this is a frequent method, it is not without flaws. The most notable of them is the fact that the Door class now relies on TimerClient. Not every type of Door necessitates the use of a timer. The original Door abstraction, in fact, had nothing to do with timing.
- If timing-free derivatives of Door are produced, those derivatives will have to supply degenerate TimeOut implementations, which could violate the LSP. Furthermore, applications that use those derivatives will have to import the TimerClient class definition, even if it isn't used.
- That reeks of Needless Redundancy and Needless Complexity.
- This is an example of interface pollution, which is prevalent in statically typed languages such as C++ and Java. Door's user interface has been contaminated with a way that it does not require.
- It had no choice but to include this method for the sake of one of its subclasses. If this technique is followed, a new method will be added to the base class whenever a derivative requires it. This will further contaminate the base class's interface, making it "fat."
- Furthermore, whenever a new method is introduced to the base class, the derived classes must implement that method. Indeed, adding these interfaces to the base class and giving them degenerate implementations is a common approach, so that derived classes are relieved of the need to implement them As we previously learned, such a technique can be in violation of the LSP, resulting in maintenance and reusability issues.

2.6.2 Separate Clients Mean Separate Interfaces

- The interfaces Door and TimerClient are utilized by completely separate clients.

TimerClient is used by Timer, and Door is used by classes that manipulate doors. Because the clients are distinct, the interfaces should be as well. Why ? Clients put pressure on the interfaces they use.

2.6.2.1 The Backwards Force Applied by Clients on Interfaces

- When we think of forces that induce software modifications, we usually consider how changes to user interfaces will affect them. For example, if the TimerClient interface changed, we would be concerned about the impact on all TimerClient users.
- However, there is force acting in the other direction. The user can sometimes compel a change in the interface.
- Some Timer users, for example, will submit several time-out requests. Take a look at the TimedDoor. It sends the Register message to the Timer when it detects that the Door has been opened, asking a timeout.
- The door closes, stays closed for a bit, and then opens again before the time-out expires. As a result, we must file a new time-out request before the previous one expires. Finally, the first timed-out request expires and the TimedDoor's TimeOut method is called. The doorbell rings erroneously.
- Using the convention provided in Listing 2 - 31, we can correct this problem. Each time-out registration includes a unique timeOutId code, which is repeated in the TimeOut call to the TimerClient. This allows each TimerClient descendant to know which time-out request is being handled.

Listing 2 - 31

Timer with ID

```
class Timer
{
    public:
        void Register(int timeout,
                      int timeOutId,
                      TimerClient* client);
};

class TimerClient
{
    public:
        virtual void TimeOut(int timeOutId) = 0;
};
```


- Obviously, this modification will affect all TimerClient users. We accept this because the absence of the timeOutId is an error that must be corrected. The design in Fig. 2.6.1, on the other hand, will make this fix effect Door and all of its clients !
- This has a strong odour of rigidity and viscosity. Why should a flaw in TimerClient affect clients of Door derivatives that don't need to be timed ? When a change in one area of the programme has an impact on other parts of the programme that are completely unrelated, the cost and ramifications of the change become unpredictable, and the likelihood of negative consequences rises considerably.

2.6.3 ISP : The Interface - Segregation Principle

- Clients should not be forced to depend on methods that they do not use.
- When clients are compelled to rely on methods they don't utilize, they are vulnerable to changes in those methods. As a result, all of the clients are inadvertently coupled.
- To put it another way, if a client relies on a class that contains methods that the client does not use but that other clients do, the modifications that those other clients impose on the class will affect the client. We want to avoid such couplings as much as possible, thus we'll segregate the interfaces.

2.6.4 Class Interfaces Vs. Object Interfaces

- Take another look at the TimedDoor. Here's an object with two different interfaces for two different clients : Timer and Door users. Because the implementation of both interfaces manipulates the same data, they must be implemented in the same object. So, how do we comply with the ISP ? When the interfaces must remain together, how can we separate them ?
- The explanation lies in the fact that clients of an object do not need to use the object's interface to interact with it. Rather, they can use delegation or a base class of the object to get access to it.

2.6.4.1 Separation through Delegation

- One option is to create a TimerClient-derived object that delegated to the TimedDoor. This answer is depicted in Fig. 2.6.2.
- The TimedDoor constructs a DoorTimerAdapter and registers it with the Timer when it wants to record a time-out request. When the Timer sends the TimeOut message to the DoorTimerAdapter, the DoorTimerAdapter forwards it to the TimedDoor.

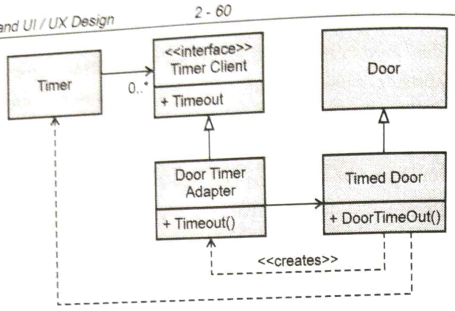

Fig. 2.6.2 Adapter for door timer

- This method complies with the ISP and prevents Door customers from being tied to the Timer. None of the users of Door would be affected if the update to Timer described in Listing 2 - 31 was implemented.
- Furthermore, TimedDoor does not have to share TimerClient's interface. The TimerClient interface can be translated into the TimedDoor interface using the DoorTimerAdapter. As a result, this is a solution that may be used for a variety of purposes.

Listing 2 - 32

TimedDoor.cpp

```
class TimedDoor : public Door
{
    public:
        virtual void DoorTimeOut(int timeOutId);
};

class DoorTimerAdapter : public TimerClient
{
    public:
        DoorTimerAdapter(TimedDoor& theDoor)
    : itsTimedDoor(theDoor)
    {}

    virtual void TimeOut(int timeOutId)
    {itsTimedDoor.DoorTimeOut(timeOutId);}

    private:
        TimedDoor& itsTimedDoor;
};
```


- This method, however, is fairly inelegant. Every time we want to register a time-out, we have to create a new object. Furthermore, the delegation needs a modest amount of runtime and memory, but it is not zero. There are application domains where runtime and memory are scarce enough to make this an issue, such as embedded real-time control systems.

2.6.4.2 Separation through Multiple Inheritances

- Multiple inheritances can be utilised to achieve the ISP, as shown in Fig. 2.6.3 and Listing 2 - 33. TimedDoor inherits from both Door and TimerClient in this model. Although the TimedDoor class can be used by clients of both base types, none is dependent on it. As a result, they use the same object via different interfaces.

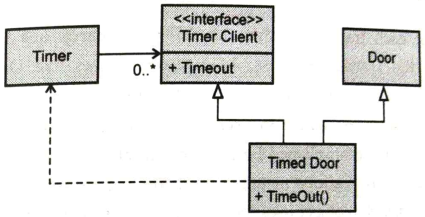

Fig. 2.6.3 Multiply inherited timed door

Listing 2 - 33

TimedDoor.cpp

```
class TimedDoor : public Door, public TimerClient
{
    public:
        virtual void TimeOut(int timeOutId);
};
```

- This is the solution I favour most of the time. Only if the translation done by the DoorTimerAdapter object was required, or if multiple translations were required at different times, would I prefer Fig. 2.6.2 over Fig. 2.6.3.

2.6.5 ATM User Interface Example

- Let's take a look at a somewhat more substantial case. The difficulty with standard Automated Teller Machines (ATMs). An ATM machine's user interface must be extremely adaptable. It's possible that the result will need to be translated into a variety of languages.

- It may be necessary to present it on a screen, a braille tablet or a speech synthesizer. Clearly, this can be accomplished by building an abstract base class with abstract methods for all of the various messages that the interface must deliver.

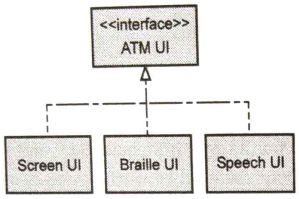

Fig. 2.6.4 ATM UI interface

- Consider that each transaction that the ATM can carry out is contained as a derivative of the Transaction class. As a result, classes like DepositTransaction, WithdrawalTransaction, and TransferTransaction may exist.
- It may be necessary to present it on a screen, a braille tablet or a speech synthesiser. Clearly, this can be accomplished by building an abstract base class with abstract methods for all of the various messages that the interface must deliver. Similarly, the TransferTransaction object uses the UI's RequestTransferAmount method to ask the user how much money he wishes to transfer between accounts. This corresponds to diagram in Fig. 2.6.5.

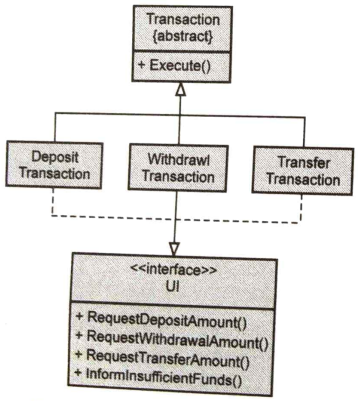

Fig. 2.6.5 Transaction hierarchy of ATM

- It's worth noting that this is exactly the circumstance that the ISP warns us about. Each transaction makes use of UI methods that aren't used by any other class.
- This raises the risk that changing one of Transaction's derivatives will compel a matching change to the UI, affecting all other Transaction derivatives and any other class that relies on the UI interface. Around here, something smells like Rigidity and Fragility.
- For example, if we were to create a PayGasBillTransaction, we'd have to implement additional UI methods to deal with the transaction's unique messages. Because DepositTransaction, WithdrawalTransaction and TransferTransaction all rely on the UI interface, they all need to be recompiled.
- Worse, if the transactions were all deployed as components in distinct DLLs or shared libraries, those components would have to be re-deployed, even if their logic remained unchanged.
- The UI interface can be separated into various interfaces such as DepositUI, WithdrawUI, and TransferUI to avoid this undesirable connection. These distinct interfaces can then be inherited in multiple ways to create the final UI interface. This model is depicted in Fig. 2.6.6 and Listing 2 - 34.

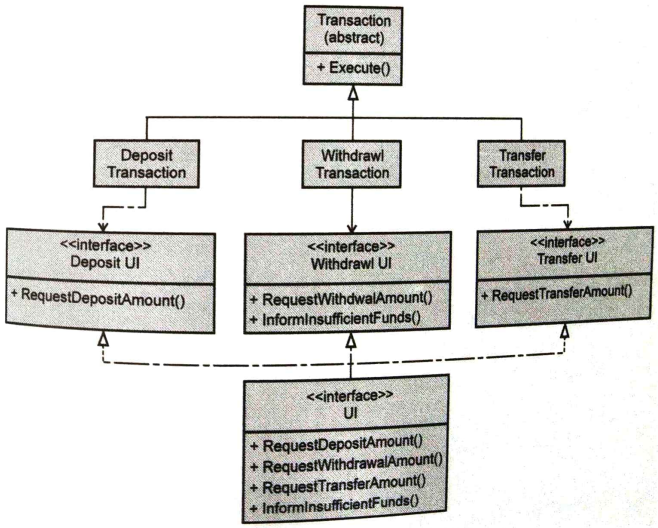

Fig. 2.6.6 Separated ATM UI interface

- A comparable base class for the abstract UI interface will be required whenever a new derivative of the Transaction class is formed; therefore the UI interface and all of its descendants must change.
- These classes, however, are not extensively utilized. They're most likely just utilized by the main process, or whatever process boots the system and builds the concrete UI instance. As a result, adding new UI base classes has a minimal impact.

Listing 2 - 34

Separated ATM UI Interface

```
class DepositUI
{
    public:
        virtual void RequestDepositAmount() = 0;
};

class DepositTransaction : public Transaction
{
    public:
        DepositTransaction(DepositUI& ui)
        : itsDepositUI(ui)
{}

virtual void Execute()
        {
        ...
        itsDepositUI.RequestDepositAmount();


        ...
        }
private:
        DepositUI& itsDepositUI;
};

class WithdrawalUI
{
        public:
                virtual void RequestWithdrawalAmount() = 0;
};

class WithdrawalTransaction : public Transaction
{
```



```
    public:
        WithdrawalTransaction(WithdrawalUI& ui)
        : itsWithdrawalUI(ui)
        {}
virtual void Execute()
        {
        ...
        itsWithdrawalUI.RequestWithdrawalAmount();
        ...
        }
private:
        WithdrawalUI& itsWithdrawalUI;
};

class TransferUI
{
        public:
        virtual void RequestTransferAmount() = 0;
};

class TransferTransaction : public Transaction
{
        public:
        TransferTransaction(TransferUI& ui)
        : itsTransferUI(ui)
        {}

        virtual void Execute()
        {
        ...
        itsTransferUI.RequestTransferAmount();
        ...
        }
        private:
        TransferUI& itsTransferUI;
};


        class UI : public DepositUI
        , public WithdrawalUI
        , public TransferUI

        {
            public:
            virtual void RequestDepositAmount();
```

```
        virtual void RequestWithdrawalAmount();
        virtual void RequestTransferAmount();
    };
```

- A close analysis of Listing 2 - 34 reveals one of the ISP compliance concerns that was not apparent in the TimedDoor example. It's worth noting that each transaction must be aware of the UI version it's using.
- DepositTransaction must be aware of DepositUI, and WithdrawTransaction must be aware of WithdrawUI, among other things. I fixed this issue in Listing 2 - 34 by requiring each transaction to be built with a reference to its own UI. It's worth noting that this enables me to use the idiom in Listing 2 - 35.

Listing 2 - 35

Interface Initialization Idiom

```
UI Gui; // global object;

void f()
{
    DepositTransaction dt(Gui);
}
```

- This is convenient, but it necessitates the inclusion of a reference element to the transaction's UI. Another option is to build a collection of global constants, as seen in Listing 2 - 36.
- Global variables aren't always an indicator of a bad layout. They have the particular benefit of being easily accessible in this scenario. It is impossible to modify them in any manner because they are references. As a result, they can't be tampered with in a way that surprises other users.

Listing 2 - 36

Seperate Global Pointers

```
// in some module that gets linked in
// to the rest of the app.

static UI Lui; // non-global object;
DepositUI& GdepositUI = Lui;
WithdrawalUI& GwithdrawalUI = Lui;
TransferUI& GtransferUI = Lui;

// In the depositTransaction.h module
```



```
class WithdrawalTransaction : public Transaction
{
    public:
        virtual void Execute()
        {
            ...
            GwithdrawalUI.RequestWithdrawalAmount();
            ...
        }
};
```

- To avoid polluting the global namespace in C++, one might be inclined to group all of the globals in Listing 2 - 36 into a single class. A good example of this is in Listing 2 - 37. However, this has an unfavourable consequence. You must #include ui globals.h in order to use UIGlobals. DepositUI.h, withdrawUI.h, and transferUI.h are all included in this.
- This means that any module desiring to utilise any of the UI interfaces must rely on all of them in some way, which is precisely the problem the ISP cautions us against. All modules that #include "ui globals.h" are required to recompile whenever any of the UI interfaces are changed. The interfaces that we had worked so hard to separate have been recombined by the UIGlobals class !

Listing 2 - 37

Wrapping the Globals in a class

```
// in ui_globals.h

#include "depositUI.h"
#include "withdrawalUI.h"
#include "transferUI.h"

class UIGlobals
{
public:
    static WithdrawalUI& withdrawal;
    static DepositUI& deposit;
    static TransferUI& transfer
};

// in ui_globals.cc
```

```
static UI Lui; // non-global object;
DepositUI& UIGlobals::deposit = Lui;
WithdrawalUI& UIGlobals::withdrawal = Lui;
TransferUI& UIGlobals::transfer = Lui;
```

2.6.5.1 Polyad Vs. the Monad

- Consider the case of a function g that requires access to the DepositUI and the TransferUI. Consider the fact that we'll be passing the user interfaces into this function. Is it appropriate to write the function prototype in this format ?

```
void g(DepositUI&, TransferUI&);
```

Or can be written as follow :

```
void g(UI&);
```

- The desire to write in the second (monadic) form is great. After all, we know that both arguments will refer to the same object in the former (polyadic) form. Furthermore, if we utilise the polyadic form, the invocation might look something like this :

```
g(ui, ui);
```

Somehow, this seems perverse.

- Whether it's perverse or not, the polyadic form is frequently preferred over the monadic form. The monadic form makes g reliant on each and every interface in UI. As a result, when WithdrawUI changes, g and all of g's clients may be affected. This is even crazier than g(ui,ui) !
- Furthermore, we can't be certain that g's two arguments always refer to the same thing ! It's possible that the interface objects will be separated in the future for whatever reason. G does not require knowledge of the fact that all interfaces have been integrated into a single object. For such functions, I prefer the polyadic form.
- **Grouping Clients**
  - The service techniques that clients use can often be used to group them together. Instead of creating separate interfaces for each client, such groupings enable for the creation of segregated interfaces for each group. This significantly minimizes the amount of interfaces the service must implement, as well as the service's reliance on each client type.
  - Occasionally, the approaches used by distinct client groups will overlap. If the overlap is minimal, the group interfaces should be kept distinct. All overlapping interfaces should define the shared functionalities. The common functions from each of those interfaces will be inherited by the server class, but it will only implement them once.

- **Changing Interfaces**
  - The interfaces to existing classes and components frequently change when object-oriented applications are maintained. Occasionally, these modifications have a significant impact and necessitate the recompilation and redeployment of a significant portion of the system.
  - Rather of modifying the existing interface, this impact can be reduced by introducing new interfaces to existing objects. Clients of the old interface can query the object for that interface to access methods of the new interface, as demonstrated in Listing 2 - 38.

Listing 2 - 38

```
void Client(Service* s)
{
  if (NewService* ns = dynamic_cast<NewService*>(s))
    {
      // use the new service interface
    }
}
```

- It's important not to overdo it, as with any principle. The prospect of a class with hundreds of different interfaces, some divided by client and others by version, would be terrifying.

2.7 Review Questions

1. *What is agile design ? List out symptoms of poor design and object oriented principle that is used to eliminate the design smells.* **(Refer sections 2.1 and 2.1.1)**
2. *Explain design smells - the odors of rotting software.* **(Refer section 2.1.2)**
3. *Agile teams don't allow the software to rot : justify with example.* **(Refer section 2.1.2.2)**
4. *Explain SRP - the single responsibility principle with example.* **(Refer section 2.2)**
5. *Explain LSP - the liskov substitution principle with example.* **(Refer section 2.4)**
6. *Explain OCP - open-closed principle with example.* **(Refer section 2.3)**
7. *Explain DIP - the dependency inversion principle with example.* **(Refer section 2.5)**
8. *Explain ISP - the interface segregation principle with example.* **(Refer section 2.6)**

❑❑❑