Object Oriented Principles

Table of Contents

1 Design Principles

1.1 What are Software Design Principles?

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The design principles are associated to Robert Martin who gathered them in "Agile Software Development: Principles, Patterns, and Practices". According to Robert Martin there are 3 important characteristics of a bad design that should be avoided:

- Rigidity - It is hard to change because every change affects too many other parts of the system.
- Fragility - When you make a change, unexpected parts of the system break.
- Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

1.2 Ten Commandments of OO Programming

Ten commandments of OO Programming as stated by Robert Martin are:

- Software entities (classes, modules, etc) should be open for extension, but closed for modification. (The open/closed principle -- Bertrand Meyer)
- Derived classes must usable through the base class interface without the need for the user to know the difference. (The Liskov Substitution Principle)
- Details should depend upon abstractions. Abstractions should not depend upon details. (Principle of Dependency Inversion)
- The granule of reuse is the same as the granule of release. Only components that are released through a tracking system can be effectively reused.
- Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed. What affects one, affects all.
- Classes within a released component should be reused together. That is, it is impossible to separate the components from each other in order to reuse less than the total.
- The dependency structure for released components must be a DAG. There can be no cycles.
- Dependencies between released components must run in the direction of stability. The dependee must be more stable than the depender.

- The more stable a released component is, the more it must consist of abstract classes. A completely stable component should consist of nothing but abstract classes.
- Where possible, use proven patterns to solve design problems.
- When crossing between two different paradigms, build an interface layer that separates the two. Don't pollute one side with the paradigm of the other.

1.3 Open Close Principle (OPC)

Software entities like classes, modules and functions should be ***open for extension*** *but* ***closed for modifications.***

OPC is a generic principle. You can consider it when writing your classes to make sure that when you need to extend their behavior you don't have to change the class but to extend it. The same principle can be applied for modules, packages, libraries. If you have a library containing a set of classes there are many reasons for which you'll prefer to extend it without changing the code that was already written (backward compatibility, regression testing). This is why we have to make sure our modules follow Open Closed Principle.

When referring to the classes Open Close Principle can be ensured by use of Abstract Classes and concrete classes for implementing their behavior. This will enforce having Concrete Classes extending Abstract Classes instead of changing them. Some particular cases of this are Template Pattern and Strategy Pattern.

1.4 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.*

Dependency Inversion Principle states that we should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. Furthermore it inverts the dependency: instead of writing our abstractions based on details, the we should write the details based on abstractions.

Dependency Inversion or Inversion of Control is a better known term referring to the way in which the dependencies are realized. In the classical way when a software module (class, framework) need some other module, it initializes and holds a direct reference to it. This will make the 2 modules tight coupled. In order to decouple them, the first module will provide a hook (a property, parameter) and an external module controlling the dependencies will inject the reference to the second one.

By applying the Dependency Inversion the modules can be easily changed by other modules just changing the dependency module. Factories and Abstract Factories can be used as dependency frameworks, but there are specialized frameworks for that, known as Inversion of Control Container.

1.5 Interface Segregation Principle

- *Clients should not be forced to depend upon interfaces that they don't use.*

This principle teaches us to take care how we write our interfaces. When we write our interfaces we should take care to add only methods that should be there. If we add methods that should not be there the classes implementing the interface will have to implement those methods as well. For example if we create an interface called Worker and add a method lunch break, all the workers will have to implement it. What if the worker is a robot?

As a conclusion Interfaces containing methods that are not specific to it are called polluted or fat interfaces. We should avoid them.

1.6 Single Responsibility Principle

- *A class should have only one reason to change.*

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handles it. When we need to make a change in a class having more responsibilities, the change might affect the other functionality of the classes.

Single Responsibility Principle was introduced by Tom DeMarco in his book “Structured Analysis and Systems Specification”, 1979. Robert Martin reinterpreted the concept and defined the responsibility as a reason to change.

1.7 Liskov's Substitution Principle

- *Derived types must be completely substitutable for their base types.*

This principle is just an extension of the Open Close Principle in terms of behavior meaning that we must make sure that new derived classes are extending the base classes without changing their behavior. The new derived classes should be able to replace the base classes without any change in the code.

Liskov's Substitution Principle was introduced by Barbara Liskov in a 1987 Conference on Object Oriented Programming Systems Languages and Applications, in Data abstraction and hierarchy

2 Open Close Principle

2.1 Motivation

A clever application design and the code writing part should take care of the frequent changes that are done during the development and the maintaining phase of an application. Usually, many changes are involved when a new functionality is added to an application. Those changes in the existing code should be minimized, since it's assumed that the existing code is already unit tested and changes in already written code might affect the existing functionality in an unwanted manner.

The **Open Close Principle** states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

2.2 Intent

Software entities like classes, modules and functions should be **open for extension** but **closed for modifications**.

2.3 Example

Below is an example which violates the Open Close Principle. It implements a graphic editor which handles the drawing of different shapes. It's obviously that it does not follow the Open Close Principle since the GraphicEditor class has to be modified for every new shape class that has to be added. There are several disadvantages:

- For each new shape added, the unit testing of the GraphicEditor should be redone.
- When a new type of shape is added, the time for adding it will be high since the developer who add it should understand the logic of the GraphicEditor.
- Adding a new shape might affect the existing functionality in an undesired way, even if the new shape works perfectly

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionality inside, written and changed by many developers, while the shape might be a class implemented only by one developer. In this case it would be great improvement to allow the adding of a new shape without changing the GraphicEditor class.

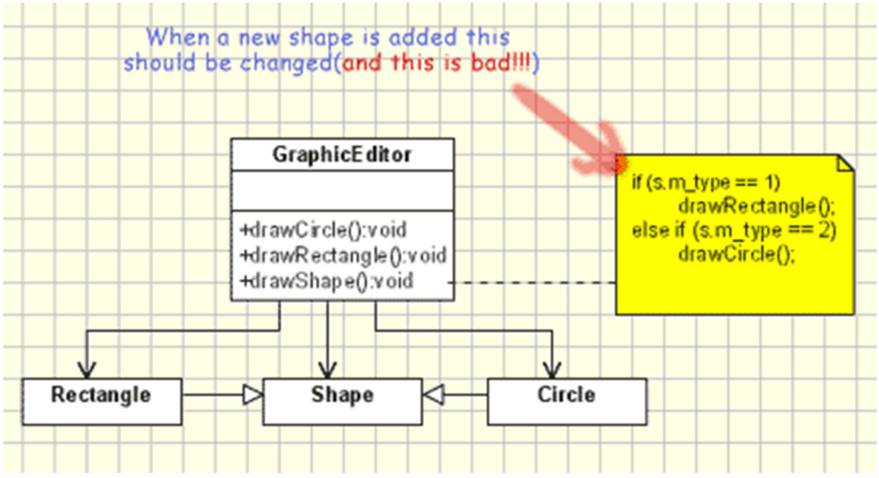

Figure 1 Open Close Prinicple - Example

```
// Open-Close Principle - Bad example
class GraphicEditor {

	public void drawShape(Shape s) {
		if (s.m_type==1)
			drawRectangle(s);
		else if (s.m_type==2)
			drawCircle(s);
	}
	public void drawCircle(Circle r) {....}
	public void drawRectangle(Rectangle r) {....}
}

class Shape {
	int m_type;
}

class Rectangle extends Shape {
	Rectangle() {
		super.m_type=1;
	}
}

class Circle extends Shape {
	Circle() {
		super.m_type=2;
	}
}
```

Below is an example which supports the Open Close Principle. In the new design we use abstract draw() method in GraphicEditor for drawing objects, while moving the implementation in the concrete shape objects. Using the Open Close Principle the problems from the previous design are avoided, because GraphicEditor is not changed when a new shape class is added:

- No unit testing required.
- No need to understand the sourcecode from GraphicEditor.
- Since the drawing code is moved to the concrete shape classes, it's a reduced risk to affect old functionality when new functionality is added.

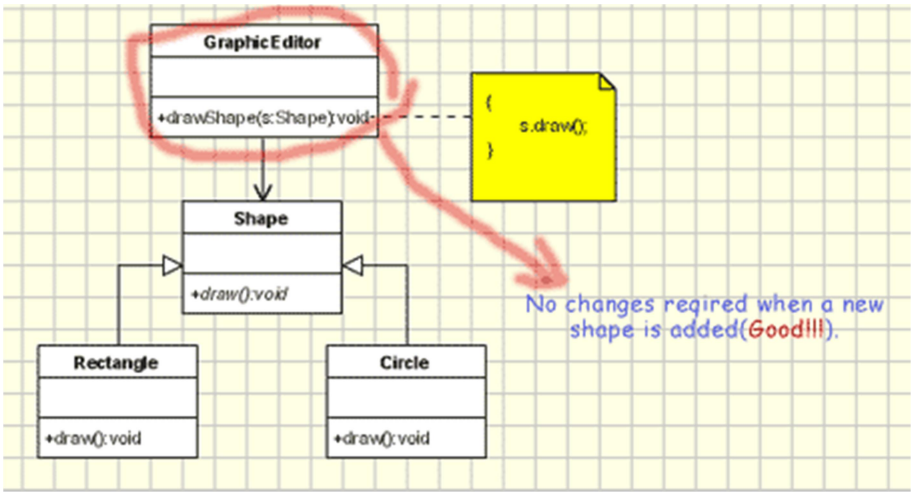

Figure 2 OPC applied

```
// Open-Close Principle - Good example
class GraphicEditor {
        public void drawShape(Shape s) {
                s.draw();
        }
}

class Shape {
        abstract void draw();
}

class Rectangle extends Shape  {
        public void draw() {
                // draw the rectangle
        }
}
```

2.4 Conclusion

Like every principle OCP is only a principle. Making a flexible design involves additional time and effort spent for it and it introduce new level of abstraction increasing the complexity of the code. So this principle should be applied in those areas which are most likely to be changed.

There are many design patterns that help us to extend code without changing it. For instance the Decorator pattern helps us to follow Open Close principle. Also the Factory Method or the Observer pattern might be used to design an application easy to change with minimum changes in the existing code.

3 Liskov's Substitution Principle (LSP)

3.1 Motivation

All the time we design a program module and we create some class hierarchies. Then we extend some classes creating some derived classes.

We must make sure that the new derived classes just extend without replacing the functionality of old classes. Otherwise the new classes can produce undesired effects when they are used in existing program modules.

Likov's Substitution Principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module.

3.2 Intent

Derived types must be completely substitutable for their base types.

3.3 Example

Below is the classic example for which the Likov's Substitution Principle is violated. In the example 2 classes are used: Rectangle and Square. Let's assume that the Rectangle object is used somewhere in the application. We extend the application and add the Square class. The square class is returned by a factory pattern, based on some conditions and we don't know the exact what type of object will be returned. But we know it's a Rectangle. We get the rectangle object, set the width to 5 and height to 10 and get the area. For a rectangle with width 5 and height 10 the area should be 50. Instead the result will be 100

```
// Violation of Likov's Substitution Principle
class Rectangle
{
        protected int m_width;
        protected int m_height;

        public void setWidth(int width){
                m_width = width;
        }

        public void setHeight(int height){
                m_height = height;
        }


        public int getWidth(){
                return m_width;
        }

        public int getHeight(){
                return m_height;
```

```
        }

        public int getArea(){
                return m_width * m_height;
        }
}

class Square extends Rectangle
{
        public void setWidth(int width){
                m_width = width;
                m_height = width;
        }

        public void setHeight(int height){
                m_width = height;
                m_height = height;
        }

}

class LspTest
{
        private static Rectangle getNewRectangle()
        {
                // it can be an object returned by some factory ...
                return new Square();
        }

        public static void main (String args[])
        {
                Rectangle r = LspTest.getNewRectangle();

                r.setWidth(5);
                r.setHeight(10);
                // user knows that r it's a rectangle.
                // It assumes that he's able to set the width and height as
for the base class

                System.out.println(r.getArea());
                // now he's surprised to see that the area is 100 instead of
50.
        }
}
```

3.4 Conclusion

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

4 Dependency Inversion Principle

4.1 Motivation

In an application we have low level classes which implement basic and primary operations and high level classes which encapsulate complex logic and rely on the low level classes. A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes. Since the high level classes are defined in terms of others this seems the logical way to do it. But this is not a flexible design. What happens if we need to replace a low level class?

Let's take the classical example of a copy module which read characters from keyboard and write them to the printer device. The high level class containing the logic is the Copy class. The low level classes are KeyboardReader and PrinterWriter.

In a bad design the high level class uses directly the low level classes. In this case if we want to change the design to direct the output to a new FileWriter class we have to change the Copy class. (Let's assume that it is a very complex class, with a lot of logic and realy hard to test).

In order to avoid such problems we can introduce an abstraction layer between the high level classes and low level classes. Since the high level modules contains the complex logic they should not depend on the low level modules and that the new abstraction layer should not be created based on low level modules. The low level modules are created based on the abstraction layer.

According to this principle the way of designing a class structure is to start from high level modules to the low level modules:

High Level Classes --> Abstraction Layer --> Low Level Classes

4.2 Intent

- 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.

4.3 Example

Below is an example which violates the Dependency Inversion Principle. We have the manager class which is a high level class, and the low level class Worker. We need to add a new module to our application because in the company there are some new specialized workers employed. We created a new class SuperWorker for this.

Let's assume that the Manager class is a complex one containing a very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:

- We have to change the Manager class (remember it is a complex one and this will involve some time and effort).

- Some present functionality from the manager class might be affected.
- Unit testing should be redone.

All those problems will take a lot of time to solve. Now it would be very simple if the application was designed following the Dependency Inversion Principle. That means that we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface. When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it.

In order to have more dramatic effect, just imagine that the Graphic Editor is a big class, with a lot of functionallity inside, written and changed by many developpers, while the a shape might be a class implemented only by one developer. In this case it would be great improvment to allow the adding of a new shape without changing the GraphicEditor class.

```
// Dependency Inversion Principle - Bad example
class Worker {
	public void work() {
		// ....working
	}
}

class Manager {
	Worker m_worker;

	public void setWorker(Worker w) {
		m_worker=w;
	}

	public void manage() {
		m_worker.work();
	}
}

class SuperWorker {
	public void work() {
		//.... working much more
	}
}
```

Below is the code which supports the Dependency Inversion Principle. In this new design a new abstraction layer is added through the IWorker Interface. Now the problems from the above code are solved:

- Manager class should not be changed.
- Minimized risk to affect old funtionallity present in Manager class.
- No need to redo the unit testing for Manager class.

```
// Dependency Inversion Principle - Good example
interface IWorker {
	public void work();
}

class Worker implements IWorker{
	public void work() {
		// ....working
	}
}

class SuperWorker  implements IWorker{
	public void work() {
		//.... working much more
	}
}

class Manager {
	IWorker m_worker;

	public void setWorker(IWorker w) {
		m_worker=w;
	}

	public void manage() {
		m_worker.work();
	}
}
```

4.4 Conclusion

When this principle is applied it means that the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer. In that case the creation of new low level objects inside the high level classes (if necessary) cannot be done using the operator new. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory and Prototype.

The Template Design Pattern is an example where the DIP principle is applied.

Of course, using this principle implies an increased effort and a more complex code, but more flexible. This principle cannot be applied for every class or every module. If we have a class functionality that is more likely to remain unchanged in the future there is no need to apply this principle.

5 Interface Segregation Principle (ISP)

5.1 Motivation

When we design an application we should take care how we are going to make abstract a module which contains several sub-modules. Considering the module implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application adding another module that contains only some of the sub-modules of the original system, we are forced to implement the full interface and to write some dummy methods. Such an interface is named fat interface or polluted interface. Having interface pollution is not a good solution and might induce inappropriate behavior in the system.

The **Interface Segregation Principle** states that clients should not be forced to implement interfaces they don't use. Instead of one fat interface many small interfaces are preferred based on groups of methods, each one serving one sub-module.

5.2 Intent

Clients should not be forced to depend upon interfaces that they don't use.

5.3 Example

Below is an example which violates the Interface Segregation Principle. We have a Manager class which represent the person which manages the workers. And we have 2 types of workers some average and some very efficient workers. Both types of workers works and they need a daily launch break to eat. But now some robots came in the company they work as well, but they don't eat so they don't need a launch break. One on side the new Robot class needs to implement the IWorker interface because robots works. On the other side, they don't have to implement it because they don't eat.

This is why in this case the IWorker is considered a polluted interface.

If we keep the present design, the new Robot class is forced to implement the eat method. We can write a dummy class which does nothing (let's say a launch break of 1 second daily), and can have undesired effects in the application (For example the reports seen by managers will report more lunches taken than the number of people).

According to the Interface Segregation Principle, a flexible design will not have polluted interfaces. In our case the IWorker interface should be split in 2 different interfaces.

```
// interface segregation principle - bad example
```

```
interface IWorker {
	public void work();
	public void eat();
}

class Worker implements IWorker{
	public void work() {
		// ....working
	}
	public void eat() {
		// ..... eating in lunch break
	}
}

class SuperWorker implements IWorker{
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in lunch break
	}
}

class Manager {
	IWorker worker;

	public void setWorker(IWorker w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

Following it's the code supporting the Interface Segregation Principle. By splitting the IWorker interface in 2 different interfaces the new Robot class is no longer forced to implement the eat method. Also if we need functionality for the robot like recharging, we create another interface IRechargeble with a method recharge.

```
// interface segregation principle - good example
interface IWorker extends Feedable, Workable {
}

interface IWorkable {
	public void work();
}

interface IFeedable{
	public void eat();
}
```

```
class Worker implements IWorkable, IFeedable{
	public void work() {
		// ....working
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Robot implements IWorkable{
	public void work() {
		// ....working
	}
}

class SuperWorker implements IWorkable, IFeedable{
	public void work() {
		//.... working much more
	}

	public void eat() {
		//.... eating in launch break
	}
}

class Manager {
	Workable worker;

	public void setWorker(Workable w) {
		worker=w;
	}

	public void manage() {
		worker.work();
	}
}
```

5.4 Conclusion

If the design is already done fat interfaces can be segregated using the Adapter pattern.

Like every principle Interface Segregation Principle is one principle which require additional time and effort spent to apply it during the design time and increase the complexity of code. But it produces a flexible design. If we are going to apply it more than is necessary it will result a code containing a lot of interfaces with single methods, so applying should be done based on experience and common sense in identifying the areas where extension of code are more likely to happens in the future.

6 Single Responsibility Principle

6.1 Motivation

In this context a responsibility is considered to be one reason to change. This principle states that if we have 2 reasons to change for a class, we have to split the functionality in two classes. Each class will handle only one responsibility and on future if we need to make one change we are going to make it in the class which handle it. When we need to make a change in a class having more responsibilities the change might affect the other functionality of the classes.

The Single Responsibility Principle is a simple and intuitive principle, but in practice it is sometimes hard to get it right.

6.2 Intent

A class should have only one reason to change.

6.3 Example

Let's assume we need an object to keep an email message. We are going to use the IEmail interface from the below sample. At the first sight everything looks just fine. At a closer look we can see that our IEmail interface and Email class have 2 responsibilities (reasons to change). One would be the use of the class in some email protocols such as pop3 or imap. If other protocols must be supported the objects should be serialized in another manner and code should be added to support new protocols. Another one would be for the Content field. Even if content is a string maybe we want in the future to support HTML or other formats.

If we keep only one class, each change for a responsibility might affect the other one:

- Adding a new protocol will create the need to add code for parsing and serializing the content for each type of field.
- Adding a new content type (like html) make us to add code for each protocol implemented.

```
// single responsibility principle - bad example

interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(String content);
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set receiver; }
```

```
        public void setContent(String content) {// set content; }
}
```

We can create a new interface and class called IContent and Content to split the responsibilities. Having only one responsibility for each class give us a more flexible design:

- Adding a new protocol causes changes only in the Email class.
- Adding a new type of content supported causes changes only in Content class.

```
// single responsibility principle - good example
interface IEmail {
        public void setSender(String sender);
        public void setReceiver(String receiver);
        public void setContent(IContent content);
}

interface Content {
        public String getAsString(); // used for serialization
}

class Email implements IEmail {
        public void setSender(String sender) {// set sender; }
        public void setReceiver(String receiver) {// set receiver; }
        public void setContent(IContent content) {// set content; }
}
```

6.4 Conclusion

The Single Responsibility Principle represents a good way of identifying classes during the design phase of an application and it reminds you to think of all the ways a class can evolve. A good separation of responsibilities is done only when the full picture of how the application should work is well understood.

7 Package Principles

7.1 Granularity

As software applications grow in size and complexity they require some kind of high level organization. The class, while a very convenient unit for organizing small applications, is too finely grained to be used

as an organizational unit for large applications. Something “larger” than a class is needed to help organize large applications.

Several major methodologists have identified the need for a larger granule of organization. Booch1, uses the term “class category” to describe such a granule, Bertrand Meyer2 refers to “clusters”, Peter Coad3 talks about “subject areas”, and Sally Shlaer and Steve Mellor4 talk about “Domains”. In this article we will use the UML 0.9 terminology, and refer to these higher order granules as “packages”.

The term “package” is common in Ada and Java circles. In those languages a package is used to represent a logical grouping of declarations that can be imported into other programs. In Java, for example, one can write several classes and incoporate them into the same package. Then other Java programs can ‘import’ that package to gain access to those classes.

7.2 Stability

Unfortunately, the volatility of a module is a difficult thing to understand. Volatility depends upon all kinds of factors. For example, some programs print their version numbers when asked. The module that contains that version number is completely volatile. It is changed every time the application is released. On the other hand, other modules are changed far less often.

Volatility also depends upon market pressures and the whims of the customers. Whether or not a module is likely to change depends upon whether that module happens to contain something that a customer is going to want to change. These things are hard to predict.

However, there is one factor that influences volatility that is very easy to measure. I call this factor stability.

The classic definition of the word stability is:”Not easily moved.” That is, stability is not a measure of the likelihood that a module will change; rather it is a measure of the difficulty in changing a module.

Clearly, modules that are more difficult to change are going to be less volatile. The harder the module is to change, i.e. the more stable it is, the less volatile it will be.

How does one achieve stability? Why, for example, are “Reader” and “Writer” so stable? Consider again the forces that could make them change. They depend upon nothing at all, so a change from a dependee cannot ripple up to them and cause them to change. This characteristic can be called “Independence”. Independent classes are classes which do not depend upon anything else.

Another reason that “Reader” and “Writer” are stable is that they are depended upon by many other classes. “Copy”, “KeyboardReader” and “KeyboardWriter” among them

In fact, the more varieties of “Reader” and “Writer” exist, the more dependents these classes have. The more dependents they have, the harder it is to make changes to them. If we were to change “Reader” or “Writer” we would have to change all the other classes that depended upon them. Thus, there is a great deal of force preventing us from changing these classes, and enhancing their stability.

Classes that are heavily depended upon can be called “Responsible”. Responsible classes tend to be stable because any change has a large impact.

The most stable classes of all are classes that are both Independent and Responsible. Such classes have no reason to change, and lots of reasons not to change.

7.3 Designing with Packages

In the UML, packages can be used as containers for a group of classes. By grouping classes into packages we can reason about the design at a higher level of abstraction. The goal is to partition the classes in your application according to some criteria, and then allocate those partitions to packages. The relationships between those packages express the high level organization of the application. But this begs a large number of questions.

1. What are the best partitioning criteria?
2. What are the relationships that exist between packages, and what design principles govern their use?
3. Should packages be designed before classes (Top down)? Or should classes be designed before packages (Bottom up)?
4. How packages are physically represented? In C++? In the development environment?
5. Once created, to what purpose will we put these packages?

To answer these questions, Robert Martin has put together several design principles which govern the creation, interrelationship, and use of packages.

7.4 List of Package Principles

The first three package principles are about package cohesion, they tell us what to put inside packages. Managing a complex project using packages and their interdependencies is one of the most powerful tools of OOD. By creating packages that conform to the following three principles listed below, we set the stage for robust, maintainable, and reusable software. Packages are the units that focus change, that enable reuse, and that provide the unit of release that prevents developers from interfering with each other.

REP	The Release Reuse Equivalency Principle	The granule of reuse is the granule of release.
CCP	The Common Closure Principle	Classes that change together are packaged together
CRP	The Common Reuse Principle	Classes that are used together are packaged together.

The last three principles are about the couplings between packages, and talk about metrics that evaluate the package structure of a system.

ADP	The Acyclic Dependencies Principle	The dependency graph of packages must have no cycles.

SDP	The Stable Dependencies Principle	Depend in the direction of stability.
SAP	The Stable Abstractions Principle	Abstractness increases with stability.

8 The Reuse/Release Equivalence Principle (REP)

THE GRANULE OF REUSE IS THE GRANULE OF RELEASE. ONLY COMPONENTS THAT ARE RELEASED THROUGH A TRACKING SYSTEM CAN BE EFFECTIVELY REUSED. THIS GRANULE IS THE PACKAGE.

Reusability is one of the most oft claimed goals of OOD. But what is reuse? Is it reusing if I snatch a bunch of code from one program and textually insert it into another? It is reuse if I steal a module from someone else and link it into my own libraries? I don't think so.

The above are examples of code copying; and it comes with a serious disadvantage: you own the code you copy! If it doesn't work in your environment, you have to change it. If there are bugs in the code, you have to fix them. If the original author finds some bugs in the code and fixes them, you have to find this out, and you have to figure out how to make the changes in your own copy. Eventually the code you copied diverges so much from the original that it can hardly be recognized. The code is yours. While code copying can make it easier to do some initial development; it does not help very much with the most expensive phase of the software lifecycle, maintenance.

I prefer to define reuse as follows. I reuse code if, and only if, I never need to look at the source code (other than the public portions of header files). I need only link with static libraries or include dynamic libraries. Whenever these libraries are fixed or enhanced, I receive a new version which I can then integrate into my system when opportunity allows.

That is, I expect the code I am reusing to be treated like a product. It is not maintained by me. It is not distributed by me. I am the customer, and the author, or some other entity, is responsible for maintaining it.

When the libraries that I am reusing are changed by the author, I need to be notified. Moreover, I may decide to use the old version of the library for a time. Such a decision will be based upon whether the changes made are important to me, and when I can fit the integration into my schedule. Therefore, I will need the author to make regular releases of the library. I will also need the author to be able to identify these releases with release numbers or names of some sort.

Thus, I can reuse nothing that is not also released. Moreover, when I reuse something in a released library, I am in effect a client of the entire library. Whether the changes affect me or not, I will have to integrate with each new version of the library when it comes out, so that I can take advantage of later enhancements and fixes.

And so, the REP states that the granule of reuse can be no smaller than the granule of release. Anything that we reuse must also be released. Clearly, packages are a candidate for a releasable entity. It might be possible to release and track classes, but there are so many classes in a typical application that this

would almost certainly overwhelm the release tracking system. We need some larger scale entity to act as the granule of release; and the package seems to fit this need rather well.

9 The Common Reuse Principle (CRP)

THE CLASSES IN A PACKAGE ARE REUSED TOGETHER. IF YOUREUSE ONE OF THE CLASSES IN A PACKAGE, YOU REUSE THEM ALL.

This principle helps us to decide which classes should be placed into a package. It states that classes that tend to be reused together belong in the same package. Classes are seldom reused in isolation. Generally reusable classes collaborate with other classes that are part of the reusable abstraction. The CRP states that these classes belong together in the same package.

A simple example might be a container class and its associated iterators. These classes are reused together because they are tightly coupled to each other. Thus they ought to be in the same package.

The reason that they belong together is that when an engineer decides to use a package a dependency is created upon the whole package. From then on, whether the engineer is using all the classes in the package or not, every time that package is released, the applications that use it must be revalidated and rereleased. If a package is being released because of changes to a class that I don't care about, then I will not be very happy about having to revalidate my application.

Moreover, it is common for packages to have physical representations as shared libraries or DLLs. If a DLL is released because of a change to a class that I don't care about, I still have to redistribute that new DLL and revalidate that the application works with it.

Thus, I want to make sure that when I depend upon a package, I depend upon every class in that package. Otherwise I will be revalidating and redistributing more than is necessary, and wasting lots of effort.

10 The Common Closure Principle (CCP)

THE CLASSES IN A PACKAGE SHOULD BE CLOSED TOGETHER AGAINST THE SAME KINDS OF CHANGES. A CHANGE THAT AFFECTS A PACKAGE AFFECTS ALL THE CLASSES IN THAT PACKAGE.

More important than reusability, is maintainability. If the code in an application must change, where would you like those changes to occur: all in one package, or distributed through many packages? It seems clear that we would rather see the changes focused into a single package rather than have to dig through a whole bunch of packages and change them all. That way we need only release the one changed package. Other packages that don't depend upon the changed package do not need to be revalidated or rereleased.

The CCP is an attempt to gather together in one place all the classes that are likely to change for the same reasons. If two classes are so tightly bound, either physically or conceptually, such that they

almost always change together; then they belong in the same package. This minimizes the workload related to releasing, revalidating, and redistributing the software.

This principle is closely associated with the Open Closed Principle (OCP) for it is “closure” in the OCP sense of the word that this principle is dealing with. The OCP states that classes should be closed for modification but open for extension. As we learned in the article that described the OCP, 100% closure is not attainable. Closure must be strategic. We design our systems such that they are closed to the most likely kinds of changes that we foresee.

The CCP amplifies this by grouping together classes which cannot be closed against certain types of changes into the same packages. Thus, when a change in requirements comes along; that change has a good chance of being restricted to a minimal number of packages.

11 The Acyclic Dependencies Principle (ADP)

THE DEPENDENCY STRUCTURE BETWEEN PACKAGES MUST BE A DIRECTED ACYCLIC GRAPH (DAG). THAT IS, THERE MUST BE NO CYCLES IN THE DEPENDENCY STRUCTURE.

Have you ever worked all day, gotten some stuff working and then gone home; only toarrive the next morning at to find that your stuff no longer works? Why doesn’t it work? Because somebody stayed later than you! I call this: “the morning after syndrome”.

The “morning after syndrome” occurs in development environments where many developers are modifying the same source files. In relatively small projects with just a few developers, it isn’t too big a problem. But as the size of the project and the development team grows, the mornings after can get pretty nightmarish. It is not uncommon for weeks to go by without being able to build a stable version of the project. Instead, everyone keeps on changing and changing their code trying to make it work with the last changes that someone else made.

The solution to this problem is to partition the development environment into releasable packages. The packages become units of work which are the responsibility of an engineer, or a team of engineers. When the responsible engineers get a package working, they release it for use by the other teams. They give it a release number and move it into a directory for other teams to use. They then continue to modify their package in their own private areas. Everyone else uses the released version.

As new releases of a package are made, other teams can decide whether or not to immediately adopt the new release. If they decide not to, they simply continue using the old release. Once they decide that they are ready, they begin to use the new release.

Thus, none of the teams are at the mercy of the others. Changes made to one package do not need to have an immediate effect on other teams. Each team can decide for itself when to adapt its packages to new releases of the packages they use.

This is a very simple and rational process. And it is widely used. However, to make it work you must manage the dependency structure of the packages. There can be no cycles. If there are cycles in the

dependency structure then the “morning after syndrome” cannot be avoided. I’ll explain this further, but first I need to present the graphical tools that the UML 0.9 uses to depict the dependency structures of packages.

Packages depend upon one another. Specifically, a class in one package may import class in a different package. This can be depicted on a class diagram as a dependency relationship between packages.

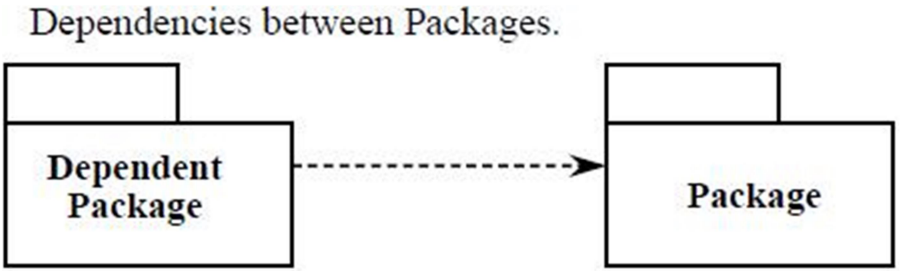

Figure 3 Package dependencies

Packages, in UML 9.0 are depicted as “tabbed folders”. Dependency relationships are dashed arrows. The arrows point in the direction of the dependency. That is, the arrow head is placed next to the package that is being depended upon. In C++ terms, there is a #include statement in a class within the dependent package that references the header file of a class in the package being depended upon.

Consider the package diagram in following Figure. Here we see a rather typical structure of packages assembled into an application. The function of this application is unimportant for the purpose of this example. What is important is the dependency structure of the packages. Notice how this structure is a graph. The packages are the nodes, and the dependency relationships are the edges. Notice also that the dependency relationships have direction. So this structure is a directed graph.

Now notice one more thing. Regardless of which package you begin at, it is impossible to follow the dependency relationships and wind up back at that package. This structure has no cycles. It is a directed acyclic graph. (DAG).

Now, notice what happens when the team responsible for MyDialogs makes a new release. It is easy to find out who is affected by this release; you just follow the dependency arrows backwards. Thus, MyTasks and MyApplication are both going to be affected. The teams responsible for those packages will have to decide when they should integrate with the new release of MyDialogs.

Notice also that when MyDialogs is released it has utterly no affect upon many of the other packages in the system. They don’t know about MyDialogs; and they don’t care when it changes. This is nice. It means that the impact of releasing MyDialogs is relatively small.

When the engineers responsible for the MyDialogs package would like to run a unit test of their package, all they need do is compile and link their version of MyDialogs with the version of the Windows package that they are currently using. None of the other packages in the system need be involved. This

is nice, it means that the engineers responsible for MyDialogs have relatively little work to do to set up a unit test; and that there are relatively few variables for them to consider.

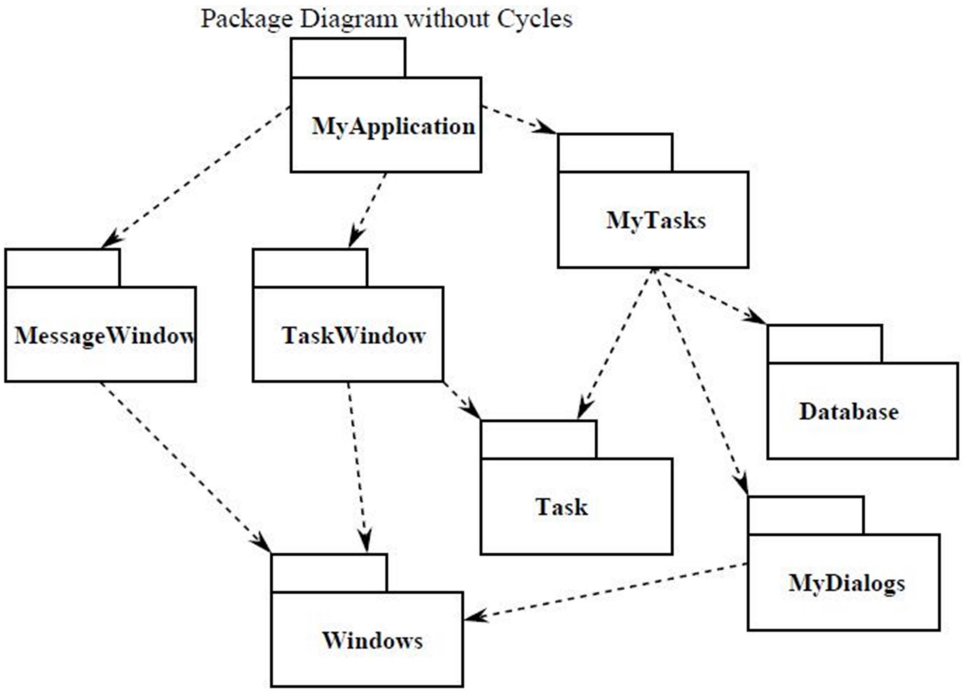

Figure 4 Packages without cycles

When it is time to release the whole system; it is done from the bottom up. First the Windows package is compiled, tested, and released. Then MessageWindow and Mydialogs. These are followed by Task, and then TaskWindow and Database. MyTasks is next; and finally MyApplication. This process is very clear and easy to deal with. We know how to build the system because we understand the dependencies between its parts.

11.1 The Effect of a Cycle in the Package Dependency Graph

Let us say that the a new requirement forces us to change one of the classes in MyDialogs such that it #includes one of the class headers in MyApplication. This creates a dependency cycle as shown in Figure.

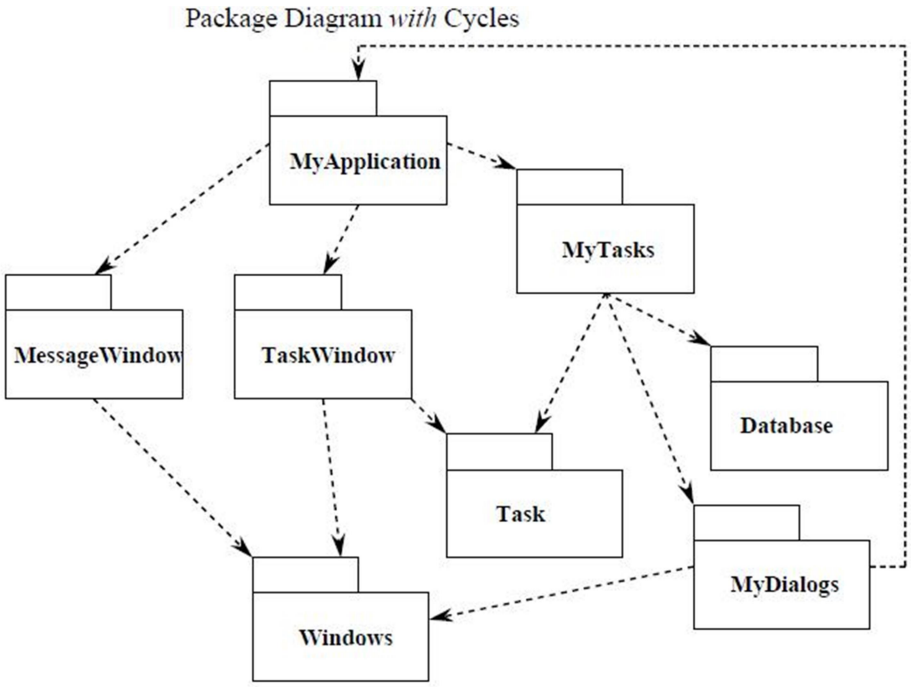

Figure 5 Packages with cycles

This cycle creates some immediate problems. For example, the engineers responsible for the MyTasks package know that in order to release, they must be compatible with Task, MyDialogs, Database, and Windows. However, with the cycle in place, they must now also be compatible with MyApplication, TaskWindow and MessageWindow. That is, MyTasks now depends upon every other package in the system. This makes MyTasks very difficult to release. MyDialogs suffers the same fate. In fact, the cycle has had the effect that MyApplication, MyTasks, and MyDialogs must always be released at the same time. They have, in effect, become one large package. And all the engineers who are working in any of those packages will experience "the morning after syndrome" once again. They will be stepping all over one another since they must all be using exactly the same release of each other.

But this is just the tip of the trouble. Consider what happens when we want to unit test the Mydialogs package. We find that we must link in every other package in the system; including the Database package. This means that we have to do a complete build just to unit test MyDialogs. This is intolerable.

If you have ever wondered why you have to link in so many different libraries, and so much of everybody else's stuff, just to run a simple unit test of one of your classes, it is probably because there are cycles in the dependency graph. Such cycles make it very difficult to isolate modules. Unit testing and releasing become very difficult and error prone. And compile times grow geometrically with the number of modules.

11.2 Breaking the Cycle

It is always possible to break a cycle of packages and reinstate the dependency graph as a DAG. There are two primary mechanisms.

1. Apply the Dependency Inversion Principle (DIP). In the case of Figure 3, we could create an abstract base class that has the interface that MyDialogs needs. We could then put that abstract base into MyDialogs and inherit it into MyApplication. This inverts the dependency between MyDialogs and MyApplication thus breaking the cycle. See Figure
2. Create a new package that both MyDialogs and MyApplication depend upon. Move the class(es) that they both depend upon into that new package.

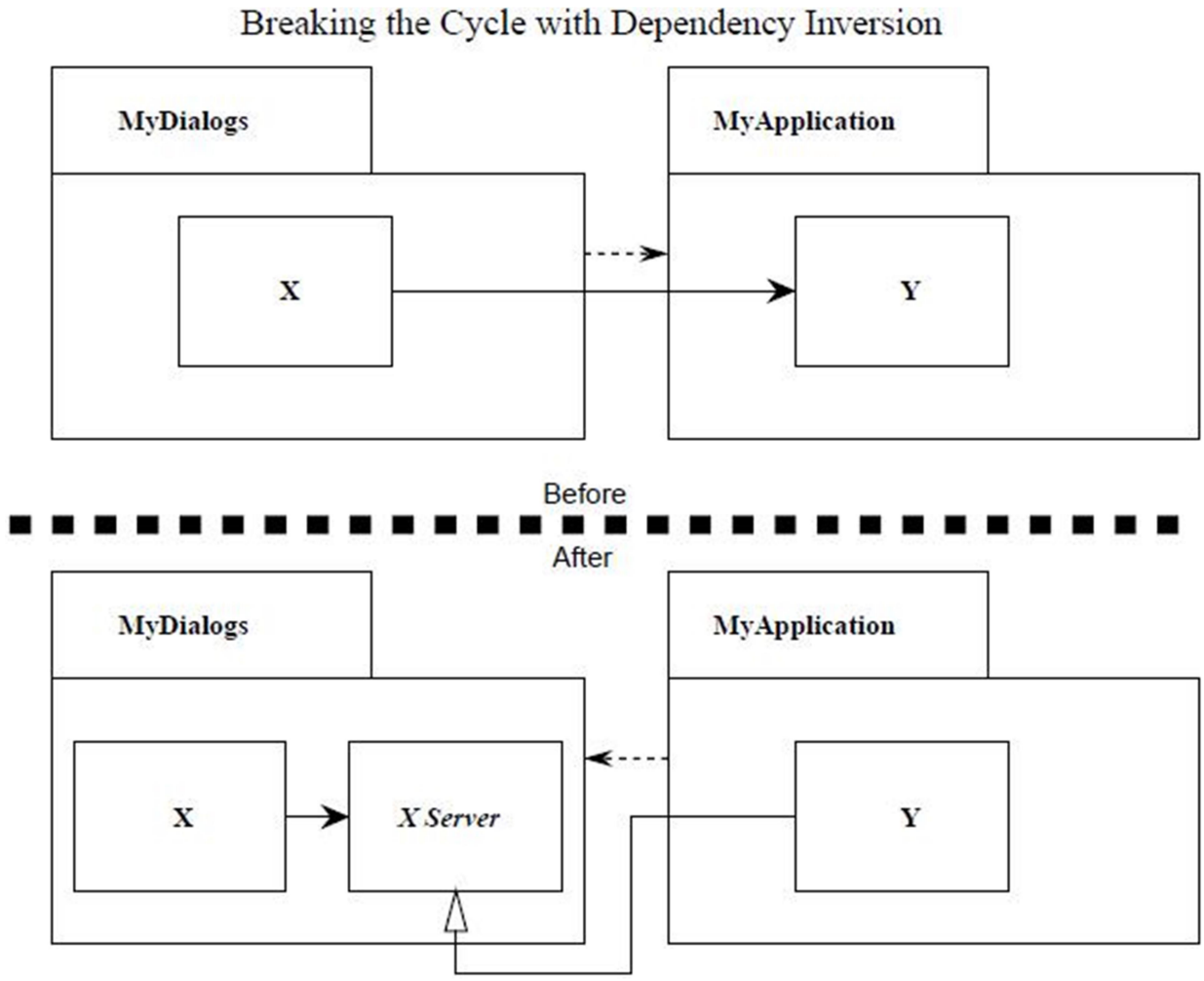

Figure 6 Breaking Cycle with DI

The second solution implies that the package structure is not stable in the presence of changing requirements. Indeed, as the application grows, the package dependency structure jitters and grows. Thus the dependency structure must always be monitored for cycles. When cycles occur they must be broken somehow. Sometimes this will mean creating new packages, making the dependency structure grow.

11.3 Top Down Design

The issues we have discussed so far lead to an inescapable conclusion. The package structure cannot be designed from the top down. This means that is it not one of the first things about the system that is designed. Indeed, it seems that it gets designed after many of the classes in the system have been designed, and thereafter remains in a constant state of flux.

Many should find this to be counterintuitive. We have come to expect that large grained decompositions are also high level functional decompositions. When we see a large grained grouping like a package dependency structure, we feel that it ought to somehow represent the function of the system. Yet this does not seem to be an attribute of package dependency diagrams.

In fact, package dependency diagrams have very little do to with describing the function of the application. Instead, they are a map of how to build the application. This is why they aren't designed at the start of the project. There is no software to build, and so there is no need for a build map. But as more and more classes accumulate in the early stages of implementation and design, there is a growing need to map out the dependencies so that the project can be developed without the "morning after syndrome". Moreover, we want to keep changes as localized as possible, so we start paying attention to the common closure principle and co-locate classes that are likely to change together.

As the application continues to grow, we start becoming concerned about creating reusable elements. Thus the Common Reuse Principle begins to dictate the composition of the packages. Finally, as cycles appear the package dependency graph jitters and grows.

If we were to try to design the package dependency structure before we had designed any classes, we would likely fail rather badly. We would not know much about common closure, we would be unaware of any reusable elements, and we would almost certainly create packages that produced dependency cycles. Thus the package dependency structure grows and evolves with the logical design of the system.

12 The Stable Dependencies Principle (SDP)

THE DEPENDENCIES BETWEEN PACKAGES IN A DESIGN SHOULD BE IN THE DIRECTION OF THE STABILITY OF THE PACKAGES. A PACKAGE SHOULD ONLY DEPEND UPON PACKAGES THAT ARE MORE STABLE THAT IT IS.

Designs cannot be completely static. Some volatility is necessary if the design is to be maintained. We accomplish this by conforming to the Common Closure Principle (CCP). By using this principle we create packages that are sensitive to certain kinds of changes. These packages are designed to be volatile. We expect them to change.

Any package that we expect to be volatile should not be depended upon by a package that is difficult to change! Otherwise the volatile package will also be difficult to change.

By conforming to the SDP, we ensure that modules that are designed to be instable (i.e. easy to change) are not depended upon by modules that are more stable (i.e. harder to change) than they are.

12.1 Stability Metrics

How can we measure the stability of a package? One way is to count the number of dependencies that enter and leave that package. These counts will allow us to calculate the positional stability of the package.

- Ca : Afferent Couplings : The number of classes outside this package that depend upon classes within this package
- Ce : Efferent Couplings : The number of classes inside this package that depend upon classes outside this package.
- I : Instability : (Ce¸ (Ca+Ce)) : This metric has the range [0,1]. I=0 indicates a maximally stable package. I=1 indicates a maximally instable package.

The Ca and Ce metrics are calculated by counting the number of classes outside the package in question that have dependencies with the classes inside the package in question. Consider the following example:

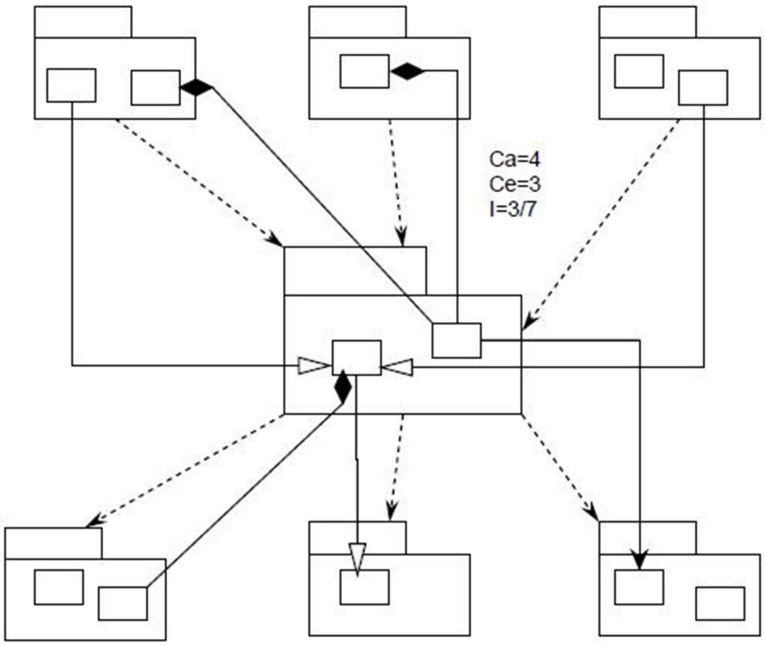

Figure 7 Stability Metrics

The dashed arrows between the packages represent package dependencies. The relationships between the classes of those packages show how those dependencies are actually implemented. There are inheritance, aggregation, and association relationships.

Now, lets say we want to calculate the stability of the package in the center of the diagram. We find that there are four classes outside that package that have relationships that target the classes inside it. Thus Ca=4. Moreover, there are three classes outside the central package that are the targets of relationships involving classes inside the central package.

Thus, Ce=3, and I=3/7.

In C++, these dependencies are typically represented by #include statements. Indeed, the I metric is easiest to calculate when you have organized your source code such that there is one class in each source file.

When the I metric is 1 it means that no other package depends upon this package; and this package does depend upon other packages. This is as instable as a package can get; it is irresponsible and dependent. Its lack of dependents gives it no reason not to change, and the packages that it depends upon may give it ample reason to change.

On the other hand, when the I metric is zero it means that the package is depended upon by other packages, but does not itself depend upon any other packages. It is responsible and independent. Such a package is as stable as it can get. Its dependents make it hard to change, and its has no dependencies that might force it to change.

The SDP says that the I metric of a package should be larger than the I metrics of the packages that it depends upon. i.e. I metrics should decrease in the direction of dependency.

12.2 Not all packages should be stable

If all the packages in a system were maximally stable, the system would be unchangeable. This is not a desirable situation. Indeed, we want to design our package structure so that some packages are instable and some are stable. The following figure shows the ideal configuration for a system with three packages.

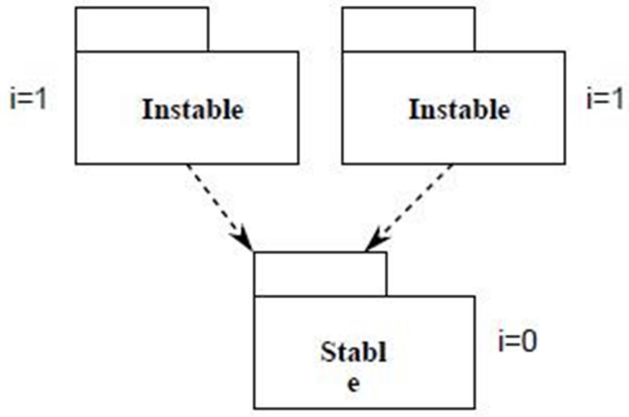

Figure 8 Not all packages to be stable

The changeable packages are on top and depend upon the stable package at the bottom. Putting the instable packages at the top of the diagram is my own convention. By arranging them this way then any arrow that puts up is violating the SDP.

Where do we put the high level design?

Some software in the system should not change very often. This software represents the high level architecture and design decisions. We don't want these architectural decisions to be volatile. Thus, the

software the encapsulates the high level design model of the system should be placed into stable packages. The instable packages should only contain the software that is likely to change.

However, if the high level design is placed into stable packages, then the source code that represents that design will be difficult to change. This could make the design inflexible. How can a package which is maximally stable (I=0) be flexible enough to withstand change? The answer is to be found in the "Open/Closed" principle. This principle tells us that it is possible and desirable to create classes that are flexible enough to be extended without requiring modification. What kind of classes conform to this principle? Abstract classes.

13 The Stable Abstractions Principle (SAP)

PACKAGES THAT ARE MAXIMALLY STABLE SHOULD BE MAXIMALLY ABSTRACT. INSTABLE PACKAGES SHOULD BE CONCRETE. THE ABSTRACTION OF A PACKAGE SHOULD BE IN PROPORTION TO ITS STABILITY.

This principle sets up a relationship between stability and abstractness. It says that a stable package should also be abstract so that its stability does not prevent it from being extended. On the other hand, it says that an instable package should be concrete since it its instability allows the concrete code within it to be easily changed.

Consider the "Copy" program. The "Reader" and "Writer" classes are abstract classes. They are highly stable since they depend upon nothing and are depended upon by "Copy" and all their derivatives. Yet, "Reader" and "Writer" can be extended, without modification, to deal with many different kinds of I/O devices.

Thus, if a package is to be stable, it should also consist of abstract classes so that it can be extended. Stable packages that are extensible are flexible and do not constrain the design.

The SAP and the SDP combined amount to the Dependency Inversion Principle for Packages. This is true because the SDP says that dependencies should run in the direction of stability, and the SAP says that stability implies abstraction. Thus, dependencies run in the direction of abstraction.

However, the DIP is a principle that deals with classes. And with classes there are no shades of grey. Either a class is abstract or it is not. The combination fo the SDP and SAP deal with packages, and allow that a packages can be partially abstract and partially stable.

13.1 Measuring Abstraction

The A metric is a measure of the abstractness of a package. Its value is simply the ratio of abstract classes in a package to the total number of classes in the package.

A = abstractClasses / totalClasses

The A metric ranges from 0 to 1. Zero implies that the package has no abstract classes at all. A value of one implies that the package contains nothing but abstract classes.

This is not a perfect metric. It presumes that a class with 20 concrete functions and one pure virtual function should be counted the same as a class with nothing but pure virtual functions. However, I have found no good way to characterize the abstractness of a class based upon a ratio of virtual to non-virtual functions. Also, the fact that a single pure virtual function exists is very significant. Imperfect though it is, I have had good results with this metric.

13.2 The Main Sequence

We are now in a position to define the relationship between stability (I) and abstractness (A). We can create a graph with A on the vertical axis and I on the horizontal axis. If we plot the two "good" kinds of packages on this graph, we will find the packages that are maximally stable and abstract at the upper left at (0, 1). The packages that are maximally instable and concrete are at the lower right at (1,0).

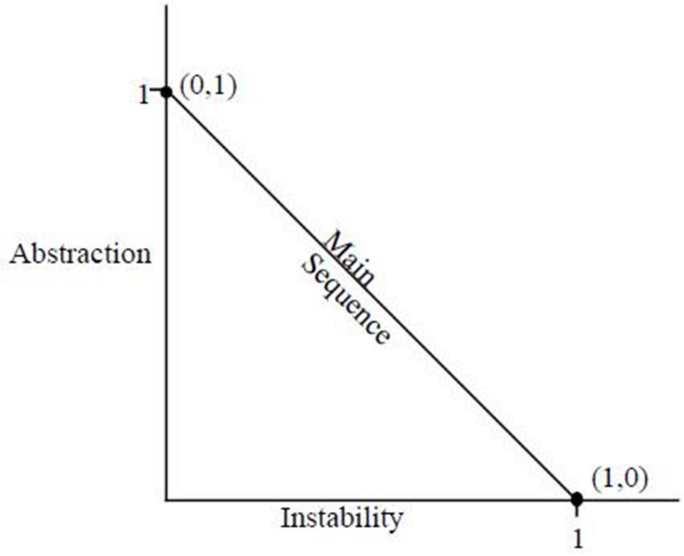

Figure 9 Abstraction Vs Instability

But not all packages can fall into one of these two positions. packages have degrees of abstraction and stability. For example, it is very common for one abstract class to derive from another abstract class. The derivative is an abstraction that has a dependency.

Thus, though it is maximally abstract, it will not be maximally stable. Its dependency will decrease its stability. Since we cannot enforce that all packages sit at either (0,1) or (1,0) we must admit that there is a locus of points on the A/I graph which defines reasonable positions for packages.

We can infer what that locus is by finding the areas where packages should not be. i.e. zones of exclusion.

Consider a package in the area of A=0 and I=0. This is a highly stable and concrete package. Such a package is not desirable because it is rigid. It cannot be extended because it is not abstract. And it is very difficult to change because of its stability. Thus, we do not want to see well designed packages sitting near (0,0). The area around (0,0) is a zone of exclusion.

It should be noted that many packages do indeed fall within the (0,0) zone. An example would be a database schema. Database Schemas are notoriously volatile and are highly depended upon. This is one of the reasons that the interface between OO applications and databases is so difficult. Another example of a package that sits on (0,0) is a package that holds a concrete utility library. Although such a package has an I metric of 1, it may in fact be non-volatile. Consider a "string" package for example. Even though all the classes within it are concrete, it may still be non-volatile. Such packages are harmless in the (0,0) zone since they are not likely to be changed.

Consider a package with A=1 and I=1. This package is also undesirable (perhaps impossible) because it is maximally abstract and yet has no dependents. It, too, is rigid because the abstractions are impossible to extend. Packages in the (1,1) zone are pretty meaningless. Thus, this too is a zone of exclusion.

But what about a package with A=.5 and I=.5? This package is partially extensible because it is partially abstract. Moreover, it is partially stable so that the extensions are not subject to maximal instability. Such a package seems "balanced". Its stability is in balance with its abstractness. Thus, the zone where A=I does not seem to be a zone of exclusion.

Consider again the A-I graph (below). We can draw a line from (0,1) to (1,0). This line represents packages whose abstractness is "balanced" with stability. Because of its similarity to a graph used in astronomy, I call this line the "Main Sequence".

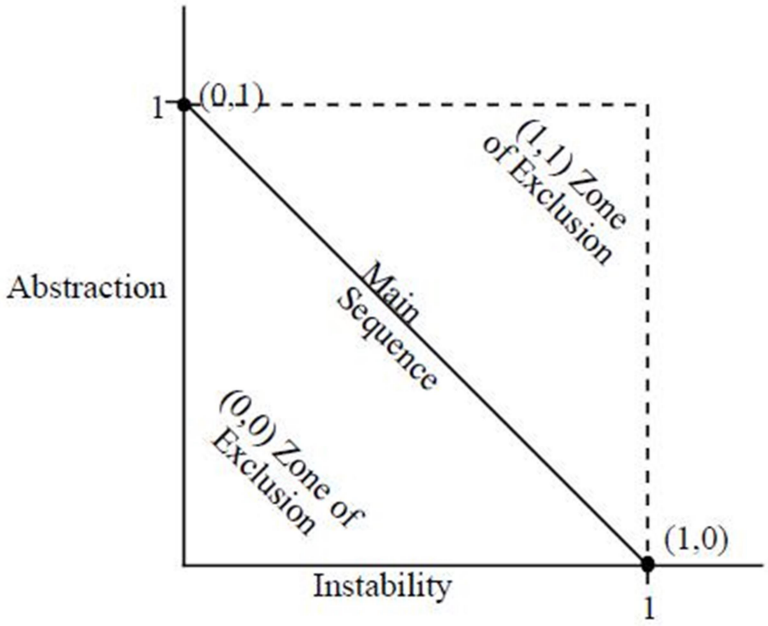

Figure 10 Zones of Abstraction Vs Instability

A package that sits on the main sequence is not "too abstract" for its stability, nor is "too instable" for its abstractness. It has the "right" number of concrete and abstract classes in proportion to its efferent and afferent dependencies. Clearly, the most desirable positions for a package to hold are at one of the two endpoints of the main sequence. However, in reality only about half the packages in a project can have

such ideal characteristics. Those other packages have the best characteristics if they are on or close to the main sequence.

13.3 Distance from the Main Sequence

This leads us to our last metric. If it is desirable for packages to be on or close to the main sequence, we can create a metric which measures how far away a package is from this ideal.

D : Distance : |(A+I-1)¸Ö2| : The perpendicular distance of a package from the main sequence. This metric ranges from [0,~0.707]. (One can normalize this metric to range between [0,1] by using the simpler form |(A+I-1)|. I call this metric D').

Given this metric, a design can be analyzed for its overall conformance to the main sequence. The D metric for each package can be calculated. Any package that has a D value that is not near zero can be reexamined and restructured. In fact, this kind of analysis have been a great aid to the author in helping to define packages that are more reusable, and less sensitive to change.

Statistical analysis of a design is also possible. One can calculate the mean and variance of all the D metrics within a design. One would expect a conformant design to have a mean and variance which were close to zero. The variance can be used to establish "control limits" which can identify packages that are "exceptional" in comparison to all the others.