**Design Principles and Design Patterns**

**Intro**

Firstly, a few points 1) before I go near any code, I’ll be describing the thinking behind all this. Do read through it all properly. 2) The code will be on the Java tip. I’ve coded substantially in C++ too but there is no support for, say, interfaces. So, it hard to port all these design patterns exactly from language to language with ease. 2) a nice book is “Java Design Patterns Essentials” by Tony Bevis. Go look that up later. 3) At the start, design patterns are not easy. You can’t learn it properly in a week. The whole thing is very abstract and you need a certain way of thinking and visualising problems. Especially if you have been taught a certain way up until now. But stick with it, as when then penny drops you will feel great, I promise! 4) Design patterns do not solve the world’s problems, there is criticism of them out there. You should read that too for a balanced view.

**Polymorphism, Binding and the Basis of Patterns**

**Polymorphism**

This is the ability of an object reference to refer to objects of different forms. Binding is evaluated at run-time rather than at compile-time. Binding is the process of connecting a message with a receiving method, I’ll talk about this very shortly.

Advantages of polymorphic languages:

1. Flexibility – they can be type checked while being flexible.

2. Abstraction – polymorphism enhances abstraction by allowing abstract operations to be defined over many types.

3. Behaviour sharing – polymorphism allows different types to share common behaviour.

4. Code sharing – code is only specified once for a range of types.

**Dynamic Binding**

We may not know what method is called until runtime. If we are using a collection of objects, at run-time the appropriate method is chosen, and this is dynamic binding. It delays the final association of a message name with a method until runtime. This is a natural way of thinking, and allows easy maintenance but it is not easy to trace code and the system must handle ‘method not found’ scenarios.

**Static Binding**

This is the simplest approach and resolves all mappings at compile time. The binding of the message to the function (if you don’t know, a method in Java is called a function in C) being called is achieved at compile time e.g. when using C or Pascal. On the upside, this means there is no runtime overhead in accessing functions/methods, and the compiler guarantees completeness as failed bindings are caught at compile time. But static bindings cannot chance without re-compilation therefore there is little support for code evolution.

**Basis of Patterns**

Design patterns centre on the combination of four OO concepts – abstraction, encapsulation, polymorphism and inheritance.

1. Abstraction – conceptualise the problem before you dive into the OO. Divide and conquer, partition things into classes and interfaces.

2. Encapsulation – objects have data, and they have methods. These are encapsulated with that said object, meaning the complexity and all the detail is hidden. We use interfaces to allow the object to communicate with other objects (the interface could just mean what public methods there are), and this allows the changeable part to be extracted, which is very common in patterns.

3. Polymorphism – this is the ability to write code that lets an object call the same method, but produce a different effect. We can work with different object types and decide on the actual object type at runtime.

4. Inheritance – where we inherit methods and data from another class. That said, design patterns tend to favour composition (‘has-a’) over inheritance (‘is-a’), discussed in the next section. Inheritance does enable polymorphism though which allows us to avoid the problems of ‘is-a’.

**SOLID Principles Overview**

Let’s now look at the SOLID principles. If you haven’t heard of these, they are five basic principles of OO that help create a system that’s easy to maintain and extend over time. They also remove what are known as ‘code smells’ by causing the programmer to refactor code until it is legible and extensible.

There are five SOLID principles:

S – Single Responsibility Principle (SRP)

O – Open Closed Principle (OCP)

L – Liskov Subsitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

I’ll explain the above later.

SOLID helps avoid degradation of design, which occurs due:

1. Changing requirements

2. Changes having to be made quickly

3. These quick changes then violate design principles, and we get an accumulation of problems over time.

4. And they are made by someone not familiar with the original design.

There are four ways to achieve SOLID:

1. Program to an interface, not an implementation – use an interface when declaring methods, which gives extra flexibility at run-time. Quick example – say we have two remote controls in our living room, one for the TV and one for a sound system. Both have a ‘menu’ button. Naturally, the menus for both will be different. So, it is better to put the ‘menu’ behaviour into an interface so we can vary the implementation. Let’s call that interface iRemote. This way we don’t need to specify which exact remote is being used at runtime, as long as it implements iRemote. So we are programming to an interface, not an implementing class.

2. Prefer object composition (has-a) over inheritance (is-a) – put object behaviours in interfaces instead of creating a big class hierarchy using inheritance. Future changes are easier therefore instead of restructuring/changing the classes in the inheritance model. Sticking with the remote control idea, say we have a TV remote with different buttons for different functions like on/off, mute, brightness, contrast, volume etc. But we can only extend one class in Java so we’d need a large hierarchy to do this using inheritance (is-a). So it is better to put the button behaviours in an interface, and have each button implement that interface, and the remote control holds references to each button – i.e. it is composed of (has-a) buttons.

3. Keep objects loosely-coupled – ideally classes should just model one thing. This is SRP, as per above. They should only be composed of other objects that are genuinely needed. If a class is copied to a new application, ideally nothing should not come with it, in other words we don’t have to copy a load of other classes over as well which the copied class is over-dependent on. If we modelled an iPhone, it has the phone and the battery. If we want to change either though, we have to change the whole unit. And if they were code classes, we would need to change the whole class, as it would contain logic for the phone functionality, and for handling the battery, which should be separate, loosely-coupled classes.

4. Encapsulate the concept that varies – consider extracting a varying part of a class into a class of its own which is referenced by the original class. That way you don’t make changes to the whole class. Quick example – say we had different types of NASA shuttle with a takeOffProcedure() method, and that implementation needed to vary at runtime, or be open to changes in future for maintenance. We should put takeOffProcedure() into a TakeOff class of its own and have each space shuttle reference it, which also means changes are only made in the TakeOff class.

**Creational Patterns**

Design patterns are comprised of main three types - creational, structural and behavioural. There are also architectural patterns, and anti-patterns. I’m going to briefly outline the creational ones now, since I think it is worth mentioning them now after what we’ve just covered. We’ll look at them, and all the others, in much more detail along with code examples you can check out.

Creational patterns provide approaches to object instantiation i.e. creation. Where you place the ‘new’ keyword in your Java code will affect how tightly or loosely coupled your classes are. There are five creational patterns:

1. Abstract Factory – provides an interface for creating families of related/dependant objects with specifying their concrete class.

2. Singleton – ensures a class has only one instance and provides a global point of access to it. For the security aware amongst you, this is global point of access can spark some debate. Though I am not going to get into that in this series.

3. Prototype – specifies the kind of objects to create using a prototypical interface, and creates new objects by copying it.

4. Factory Method – creates objects without specifying the exact class of object that will be created.

5. Builder – separates the construction steps of a complex object from its representation.

**Code Smells**

Here are seven code or design smells:

1. Rigidity – if we make one change, that ripples through the code causing other changes

2. Fragility – if we make one change, the software breaks in many places

3. Immobility – the effort/risk of separating reusable code is too great

4. Thickness – hacks are easier to implement than solutions that adhere to design (we’ll all done this…)

5. Needless complexity – too many abstractions in an effort to avoid the other six smells in this list!

6. Needless repetition – copying and pasting code throughout the system

7. Opacity – the code is just hard to understand

**SOLID Principles in Detail**

Ok, deep breath, let’s get into this then…here are the SOLID principles in detail. Afterward there are example violations complete with UML and, for the first time, a little (sometimes pseudo) Java code.

**S – Single Responsibility Principle (SRP)**

A class should only have one reason to change, or one responsibility. A responsibility is a reason for change. Not applying this leads to rigidity and fragility, two of the code smells. Though be mindful of how far to take it. SRP is conducive to high cohesion as it reduces module complexity, increases reusability and maintainability.

**O – Open Closed Principle (OCP)**

A design smells of rigidity when a change results in cascading changes to dependent modules. OCP says refactor, so more change does not mean more modification. Good OCP means good changes happen by adding code, not changing old code. Modules conforming to OCP are Open for extension and Closed for modification i.e. OCP. Behaviour is extended via interfaces/abstract classes (literally “programming to an interface”). Subtypes of the interface can be created but the interface must stay the same. We can also use method overriding for modification. Note when conceptualising, we can’t predict all changes, we can only predict likely ones - being open to all possibilities is wasteful. Test-Driven-Development (TDD) helps identify changes early as it forces us to abstract when we test. I won’t talk about TDD in this series however it is good practice to have short development cycles, do the most important features first, and have early and frequent releases.

**L- Liskov Substitution Principle (LSP)**

Barbara Liskov came up with this (ex-MIT, an ACM Turing award winner, pretty cool!) LSP checks if inheritance (is-a) is used correctly or not. Subtypes must be substitutable for their base types. Not adhering to it causes rigidity. If a subtype reference is passed into a function and it misbehaves or gives bad output, the reference violates LSP. We can’t change code to validate the reference and then call the function as this will violate OCP, like if we used an if statement to check subclasses being passed in - functions that use references to base classes must be able to use objects of subclasses without knowing it.

Is-a relationships are about behaviour – the author of a class explicitly states the contract for that class. Bertrand Meyer’s (Design by Contract) states “a routine re-declaration in a derivative may only replace the original pre-condition by one equal or weaker, and the original post-condition by one equal or stronger.” Heeding this advice, factor out common behaviour as an interface/abstraction instead of deriving and creating subclasses. Factoring makes evolution easier by creating a common abstract superclass and moving common responsibilities to it. Later extensions can then add new subclasses.

A precondition of a class is a rule that must be in place before an action can be taken. For example, before calling a method that reads from a database you may need to satisfy the precondition that the database connection is open. Postconditions describe the state of objects after a process is completed. For example, a database connection is closed after executing a SQL statement. The LSP states that the preconditions of a base class must not be strengthened by a subclass and that postconditions cannot be weakened in subclasses.

**I – Integration Segregation Principle (ISP)**

Clients should not be forced to depend on methods they do not use, they should only depend on methods they call. If they are forced to depend on methods they do not use, they’re subject to changes in those methods, meaning inadvertent coupling, as other clients can force changes on those interfaces and methods. ISP concerns interfaces that are too fat with big groups of methods where different clients use different ones. Non-cohesive interfaces are a necessary evil sometimes, but clients should not see them as a single class. Better to use interfaces that are or appear to be cohesive. E.g. if an interface had functions A, B and C in it for use by client 1, 2 and 3, it’s better to segregate them so there is a separate interface for A, B and C. Separate clients mean separate interfaces to avoid coupling, and to avoid forcing changes. Interface pollution leads to unnecessary implementation of methods and unnecessary complexity, another one of our code smells.

There are two ways to separate interfaces:

1. Delegation – I’m going to briefly mention a pattern here that we’ll look at later. Delegation uses the adapter pattern to create an adapter object that derives from a client interface and delegates messages back to a subclass. Each delegation needs a new object though, so be mindful of memory.

2. Multiple Inheritance - implement an interface and extend a superclass to incorporate required behaviours, instead of having one big fat interface.

Here’s an example. Say we had an ATM system with a really fat UI interface containing all the different transaction methods, and then each transaction class implemented that interface. In such a case, ISP is violated as each class has to implement all the transaction methods, even the unnecessary ones they don’t use. The solution is to add more interfaces to segregate the original fat UI interface, and in each interface declare only the methods a given transaction class (aka client) invokes. Then each client only implements the required methods from the relevant interface. So, all the clients are independent of each other, which is better for reuse and means less rework after a change.

**D – Dependency Inversion Principle (DIP)**

This one is tricky to visualise but I’ll do my best to explain. Traditional procedural programming (doing something to something else) creates a dependency structure that is vulnerable to changing details. OO inverts this dependency structure as details and policies depend on abstraction, and interfaces are owned by their clients. If dependencies are not inverted, it is a procedural design. DIP is fundamental low-level mechanism behind many OO benefits.

What that all means is that high level modules should not depend on low-level modules. Both should depend on abstractions. And abstractions shouldn’t depend on details, details should depend on abstractions. Traditionally high level modules with important business rules depend on lower ones, and policy depends on detail. It’s better to use inversion so the module that handles the business rules depends on interfaces/abstract base classes. These high level modules should be independent, making them easier to maintain and reuse. Avoid layering with transitive dependency - call services using abstract interfaces instead so the upper layers don’t depend on lower layers and so there are no direct inter-layer dependencies.

This is an example of the Hollywood Principle – “don’t call us, we’ll call you!”. The interface is owned by the client and packaged with it. Since the high level modules contain the complex logic, they should not depend on the low level modules, so the new abstraction layer should not be created based on low level modules. Low level modules are to be created based on the abstraction layer. When DIP is applied it means the high level classes are not working directly with low level classes, they are using interfaces as an abstract layer, and instantiation of new low level objects inside the high level classes cannot be done using the operator new.

For volatile classes where values may change between accesses, they should depend on abstractions i.e. all relationships in a program should terminate on an abstract class or interface. Ideally no variable should hold a reference to concrete class, no class should derive from concrete class, and no method should override an implemented method of any of its base classes.

**Example Violations**

**SRP – Single Responsibility Principle**

****Not applying SRP leads to rigidity and fragility. Below, the Rectangle class has two responsibilities, or two reasons to change – remember it should only have one.

And, in an unrelated code example, the Modem interface has two responsibilities also, again it should only have one.

public interface Modem

{

public void Dial(string phoneNum); // connection responsibility

public void Hangup();

public void Send(char c); // communication responsibility

public char Receive();

}

**OCP – Open Closed Principle**

Not applying OCP leads to rigidity. Good OCP is where we add new code, not change old code. Below, there is a fixed abstraction to decouple the client from the server, ClientInterface. Again, programming to an interface. Then we can add different server classes and ClientInterface stays the same. Another way is to split out the changeable bit into a separate method in an abstract class, then it can be overridden if needed.

Before…. And with the refactoring….





In this shape-drawing example, imagine to begin with we only cater for squares and circles. If we want to add triangles we need to modify existing code to include a triangle in both drawing and decision logic. To solve this, encapsulate the concept that varies, put an abstract draw method in a shape interface, and pass objects of type ShapeType into the relevant draw method in DrawingTool. Then we can add more shapes easily.

**LSP – Liskov Substitution Principle**

Not applying LSP leads to rigidity. A useful heuristic is factoring instead of deriving, meaning, as we’ve discussed plenty of times so far, factoring out common behaviour as an abstraction/interface. Remember subtypes must be substitutable for their base types.

public class DrawingTool {

public void drawShape(Shape s) {

if (s.type == ShapeType.CIRCLE) {

System.out.println("I'm drawing a circle");

} else if (s.type == ShapeType.SQUARE) {

System.out.println("I'm drawing a square");

} else {

System.out.println("It’s not a square or a circle - what do I do?");

}

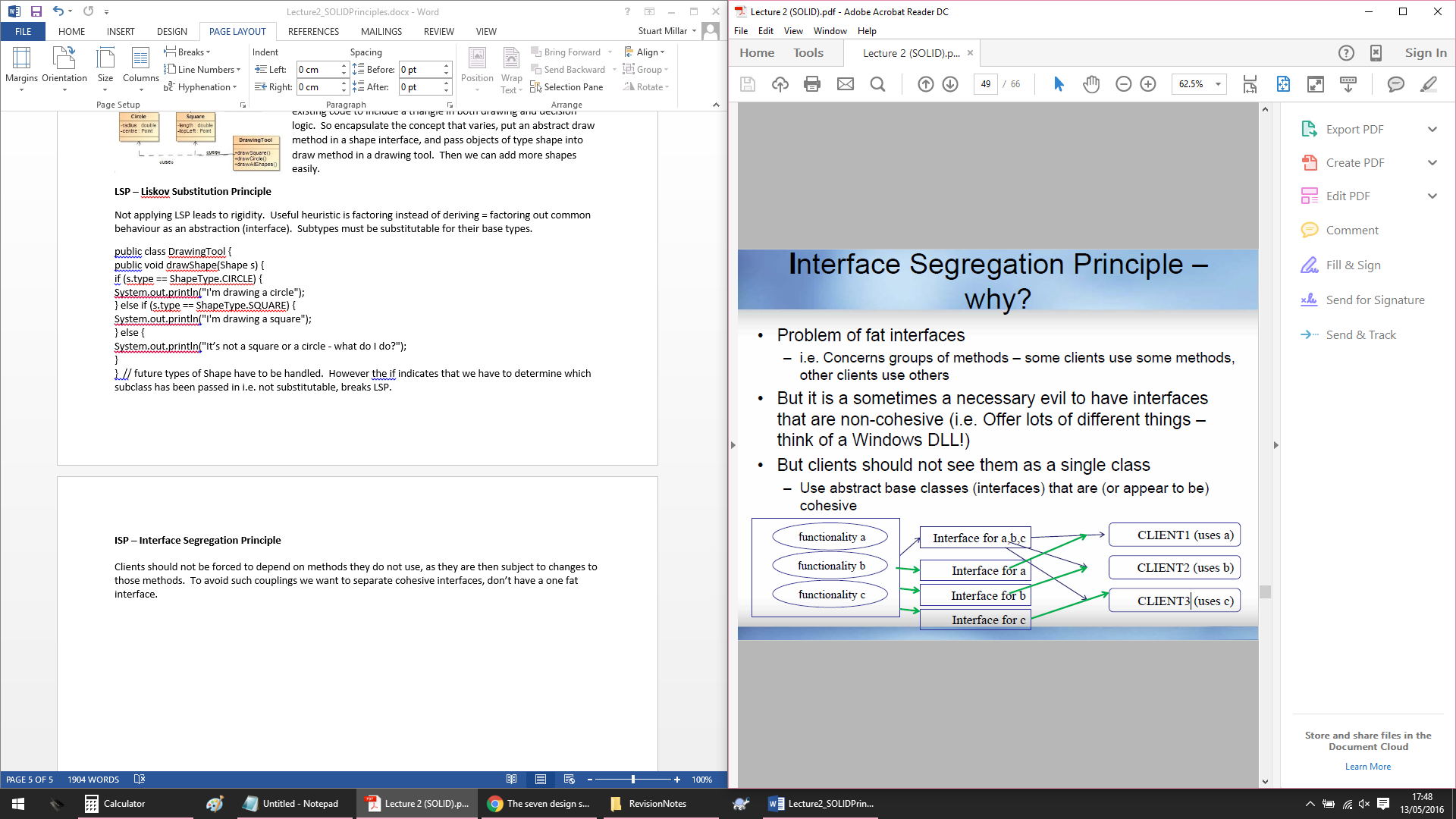
}

}

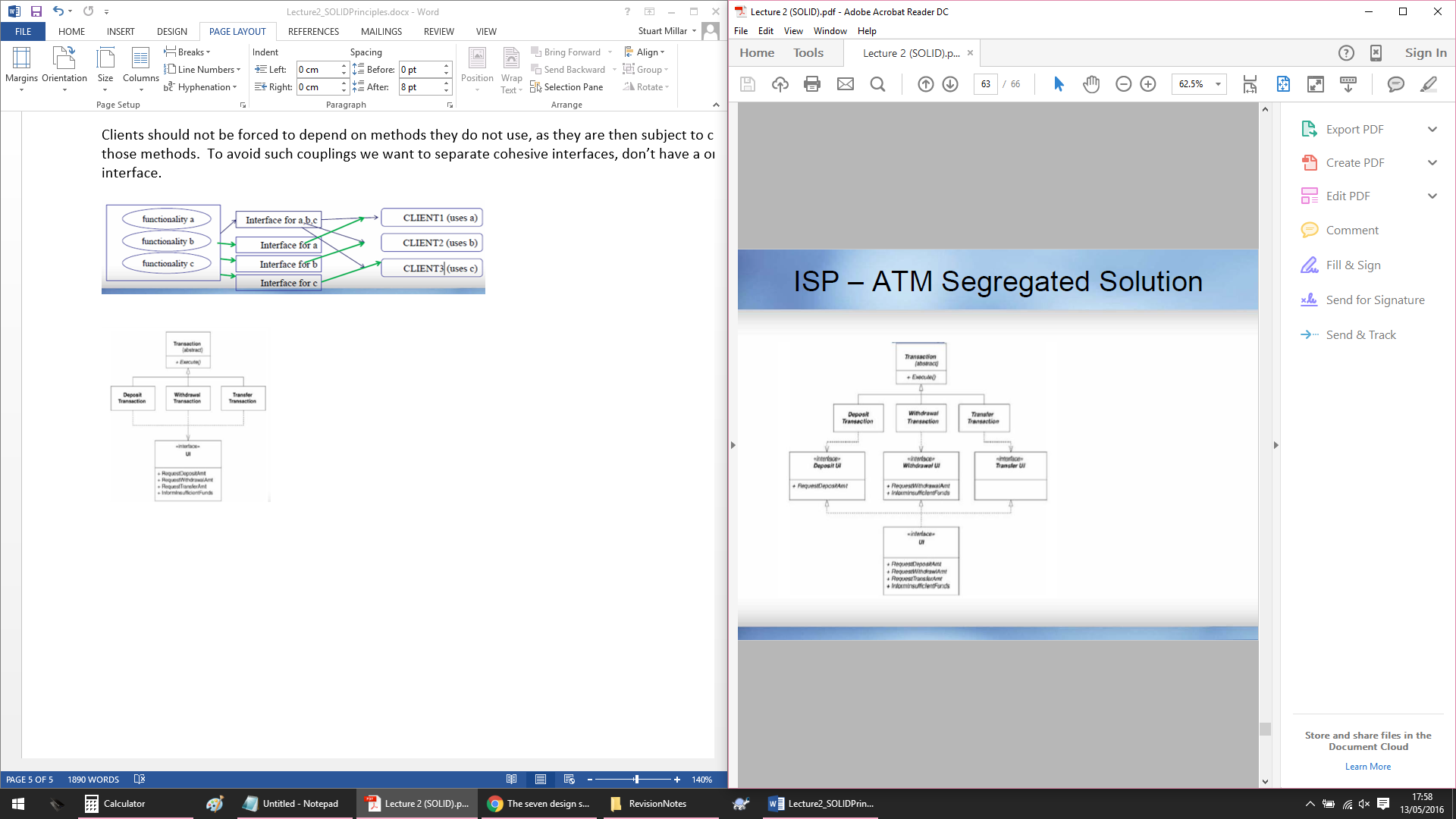
Above, future types of Shape have to be handled. However, the if statement indicates that we have to determine which subclass has been passed in i.e. they are not substitutable, breaking LSP. And what if we need a triangle?

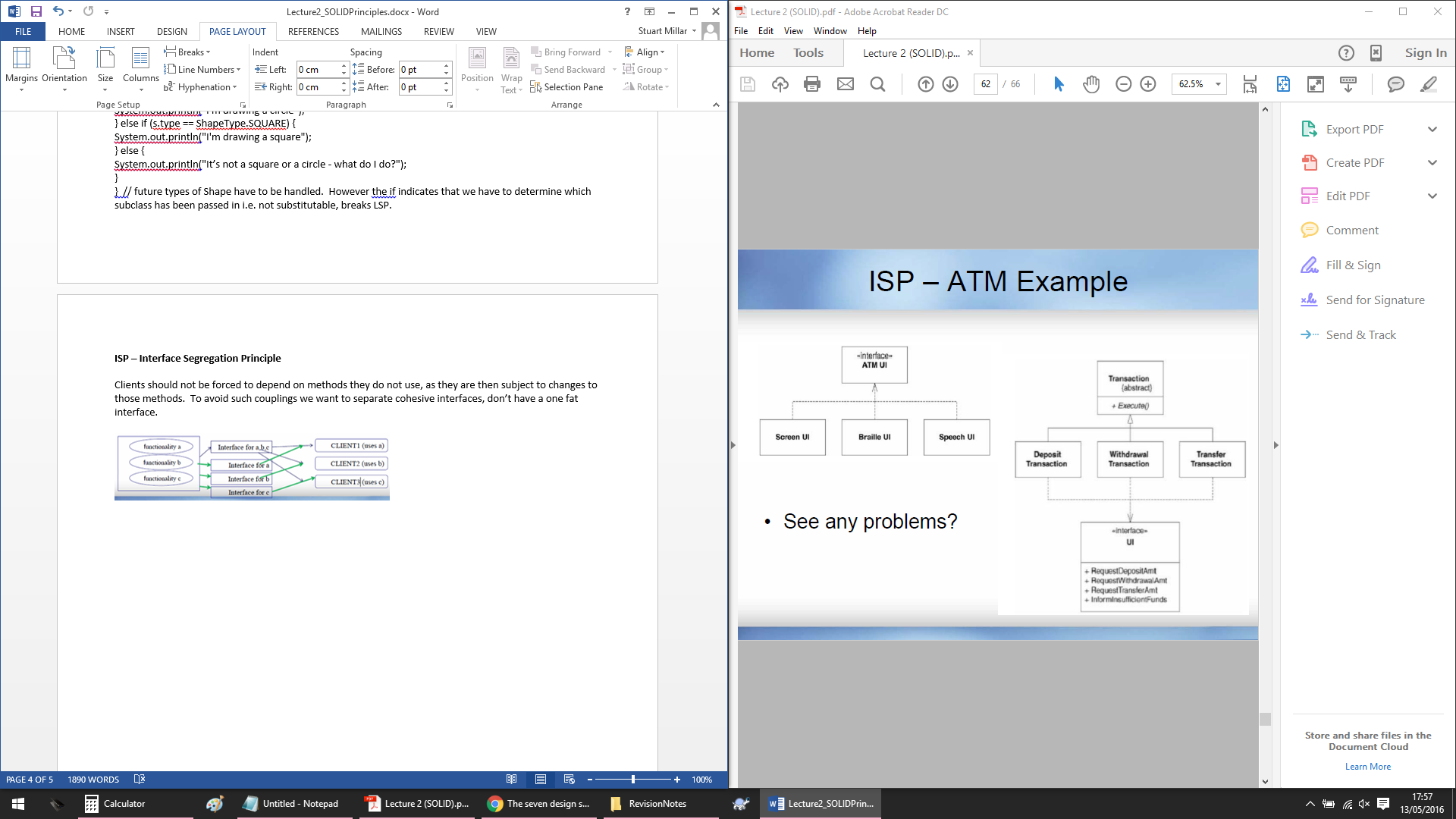
**ISP – Interface Segregation Principle**

Clients should not be forced to depend on methods they do not use, as they are then subject to changes to those methods. To avoid such couplings, we want to separate cohesive interfaces, instead of one fat interface.



Recall the ATM system example from earlier. Here is a before, and after refactoring…





**DIP – Dependency Inversion Principle**

High-level modules should not depend on low-level modules, both should depend on abstractions. And abstractions should not depend on details, details should depend on abstractions. Instead, invert and provide abstract interfaces so there are no dependencies between high and low-level modules.

Before…

****

And after refactoring…

