# Chapter one. Single responsibility principle.

This is the first of the five principles. What is it?

The idea of the single responsibility principle is that classes or modules only have a single reason to change. The word reason might be confusing here.

A class can do more than one thing obviously. This principle is not about keeping classes small. The goal here is to prevent a class from changing all the time for all sorts of reasons.

I'll give you an example. Class one contains functionality A and functionality B. If you change A, you can break B just from the fact that they share the same class.

A black rectangular object with letters

Description automatically generated

You have seen these kinds of problems, big classes with lots of code are intimidating. When you change something, you just have to hope you did not break something else. But there are more problems.

Let me give you a list of them.

1. Classes get big when they have many responsibilities. Naming the class is hard and finding code is hard.
2. Mixing responsibilities. We will look at an example of an employee class that mixes responsibilities and find out it is dangerous to change code around unrelated other codes.
3. Dependencies on libraries. I will give you an example, where a harmless dependency in one class starts to invade another unrelated class.

Let's start with a common example of the single responsibility principle and how it is violated. It concerns a class that does too many different things.

Look at the employee class. It stores a name and salary.

A screenshot of a computer program

Description automatically generated

The employee class also has some business logic to raise the salary. Until here, we all consider this to be business logic.

But the class can also save an employee as XML. And print a report of the employee. The name of the class implies an employee entity, but the class does more than that. It also handles storage and reporting. You would not expect this by just looking at the class name.

The class name should tell us what it is responsible for.



If you see big class names like this, it might be the first clue that the class is doing too many different things. As you see, naming is hard, but also finding code in such classes is hard. And our problems are only starting to get bigger from here. Clearly, the employee class has mixed responsibilities. Let's look at a practical problem that arises.

Look at the employee class again. For this example, I have removed the reporting function.

A screenshot of a computer code

Description automatically generatedEmployee now has two responsibilities: Business logic and storage logic. I implemented the save\_as\_xml method. It opens the file emp.xml and writes XML to it. Look at the file name.

It is hardcoded. To make it more flexible for users of the class, the file name will be defined as a class variable at the top of the class. The variable is created. And used in the function.

Notice how the code for the two responsibilities is now scattered all over the class. And there is a new subtle problem.

A computer code with text and numbers

Description automatically generated with medium confidence

The class variable with the file name is only used by the storage part. User of this class start reading the code from the top and ask themselves, why does this class know anything about XML file names?

The user now has to scan the code to find out it has nothing to do with the business logic, but with the later added storage responsibility.

A screenshot of a computer code

Description automatically generated

This problem will get worse when we change the class again. At this point, the employee can be saved as XML, but now we want to replace XML with JSON.



The first thing we do is remove the save\_as\_xml function.

A screenshot of a computer code

Description automatically generated

That cleans up the code but wait a minute. We forgot the class variable xml\_filename. Imagine someone looking at this code in six months. Will this person be brave enough to delete the variable?

A screenshot of a computer code

Description automatically generated

You see that when responsibilities are mixed, it is not only harder to add new code, but you can also not safely delete code and be sure you have not forgotten something.

Look at the employee class diagram. We have removed the save\_as\_xml method but need to replace it with the save\_as\_json method.

A black and white box with text

Description automatically generated

A white rectangular box with black text

Description automatically generated

But if we add the save\_as\_json method, it will cause the same problems as before. Clearly, we are violating the single responsibility principle.

A close-up of a sign

Description automatically generated

I will show you even more violations, but before I do, I'd like to show you how the problem is solved. We know that the class is doing too many things, and we need to split it up into multiple classes.

First, we identify the responsibilities.

A black and white rectangular box with black text

Description automatically generated

The employee class contains business logic and storage logic. The storage logic will be moved to its own class.

A couple of rectangular boxes with black text

Description automatically generated with medium confidence

When classes are split up, new problems are introduced: Missing dependencies. The save\_as\_json method is now in the EmployeeStorage class.

How does it get access to an employee object?

A black arrow pointing to a white rectangular object

Description automatically generated with medium confidence

For now, We choose the simplest solution. We pass it when we call the save\_as\_json function.

Here is the employee storage class.

A computer code with text

Description automatically generated with medium confidence

One thing is immediately clear. Every line of code in this class has something to do with employee storage.

The employee class is stripped of storage logic and only contains business logic.

A white background with black text

Description automatically generated

And how do we connect both classes?

We do this in main.py.

A white background with black text

Description automatically generated

Main imports both classes.

An employee object is created, followed by an employee storage object. And finally, the save\_as\_json method is called and an employee object is passed to the function.

A white background with black text

Description automatically generated

Here is a dependency diagram of the program we just created.

A diagram of a company

Description automatically generated

As I said before, OO allows us to manage our dependencies and the S.O.L.I.D. principles use this kind of dependency management. We will discuss dependency management further in the dependency inversion principle chapter. But now we continue with the single responsibility principle.

At this point, we have achieved the single responsibility in both classes.

A screenshot of a computer program

Description automatically generated

But we have to be careful because we can violate the principle again very easily.

Let me give you another example. Sometimes we violate the single responsibility principle in very subtle ways.



In the following example, extra responsibilities sneak in through an external library.

Look at the save\_as\_json method.

A computer code with green and blue text

Description automatically generated

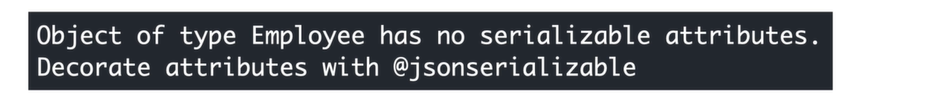
The method constructs the JSON manually, and this is error-prone. You found a library online that serializes and saves JSON in a single line of code.

All we need to do is import jsonlibrary. And call jsonlibrary.save.

A screenshot of a computer code

Description automatically generated

We run the program and get an error.



The JSON library tries to serialize the employee object. To do this, it searches for serializable attributes. But we have not marked the name and salary as serializable.

No problem. The error shows that all we need to do is to decorate the attributes with @jsonserializable.

Let's do it now.

I add the two decorators and a question rises.

Where do the decorators come from?

A screenshot of a computer code

Description automatically generated

The answer is that they come from the JSON library. That means we have to import jsonlibrary in the employee class. And this makes it very clear. Our goal was to get rid of all storage functionality in the employee class and suddenly the employee is polluted with knowledge about JSON serialization. Think about the complications here, it becomes impossible to move the employee entity to another project without installing the JSON library here. And whenever JSON serialization becomes obsolete, we need to make changes to the employee class, which should be unaware of all things related to JSON.

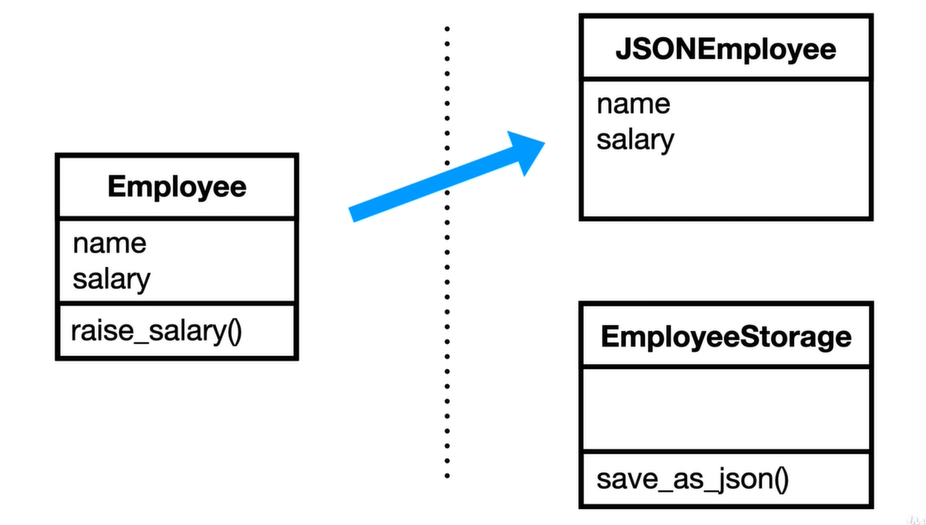
How would you solve a problem like this?

There is only one way to solve it without breaking the single responsibility principle. On the left, you see the employee class that should not know anything about JSON. On the right, there is the employee storage class that needs a serializable employee object. The boundary in the middle separates the entities from the storage logic. A new class has been created: JSONEmployee.

A close-up of a table

Description automatically generated

It is also hosted in the storage module. Its only purpose is to provide a copy of the employee data attributes and mark them JSON serializable. Each data attribute is copied from the employee object to the JSONEmployee object.



Then the JSONEmployee object is passed to the save\_as\_json method.

A screenshot of a computer

Description automatically generated

Something similar happens for de-serializing.

A diagram of a diagram

Description automatically generated

I've worked on many projects where colleagues protested against this extra object in the middle. I understand them because it seems to be a lot of extra work when all the data attributes for a lot of objects need to be copied. But if you want to respect the single responsibility principle, the employee object must not know about JSON.

There are more examples of violations like this. You'll find them in object relational mapper frameworks. In such frameworks, entities get multiple responsibilities.

* Like storing data.
* Provide UI labels.
* Validate data.
* And provide error messages.

Let me show you an example of such classes. The employee class starts as a business object to store employee data. But suddenly it is also responsible for labels on the screen. And even error messages on the screen. This class is now responsible for many things and violates the single responsibility principle.

A screenshot of a computer error

Description automatically generated

**Conclusion:**

A hand with a thumb up

Description automatically generatedWhen the single responsibility principle is violated, your classes get bigger, and it gets riskier to change code without affecting other code. Sometimes more subtle violations happen, like in the decorator examples. Keep an eye on the import statements in your classes; they can provide clues for these kinds of violations.

# Chapter two. Open-closed principle.

This principle causes a lot of confusion for developers who first

hear it.

Open-closed means open for extension, but closed for modification.

That might even make it more confusing.

How can a program be extended if you cannot change it?

Surely you have to modify existing code to add new functionality.

Yes, but if you stick to this principle, the changes do not explode into all directions.

Perhaps you recognize code like this. This is a selection that switches on object types. The reason

to use the open-closed principle is easy.

It prevents switches like this.

Why is this important?

First, I'll show you how a selection on types violates the open closed principle and how to solve

it.

Then I show how easy it is to add new functionality when we adhere to the principle.

Finally, I will remove existing functionality to show how we can do it with the least amount of risk.

Let me start by showing you problematic code.

Here is a base class called Employee, it stores an employee name.

Look at the print\_employee function, it takes a reference to an object of type employee. It prints the

name of the employee.

So far, so good.

Now the system needs to be extended with a manager class.

A manager inherits from employee and adds an attribute: department.

Here is the code that adds the manager class.

Manager inherits from employee and adds an extra attribute to the object: department.

I overload the initializer to support the department attribute.

At this point, e can be an object of type employee or manager.

We know that e has a name, but we don't know if e has a department.

The print\_employee function needs a switch to check the type of e.

In order to check for the type, the switch needs to know all possible employee types.

When the print\_employee function is moved to a different module, this problem becomes even more visible

because to know every employee type, they need to be imported.

And this switch can be in multiple places. Every time a new employee subclass is created, you have

to change the code in all those places.

It now becomes very clear that we cannot extend the functionality without changing existing working

code.

And there is the risk of changing the code in one place, but forget it in another place.

This example is a violation of the open-closed principle.

The good news is that it is a solvable problem. And the technique to solve the problem is polymorphism.

The employee will get a new method called get\_info that returns employee information.

The manager will override this method to return manager information.

This is how the code looks like.

get\_info in the employee class returns the employee name.

get\_info in the manager class returns the name plus department.

The print\_employee function does not have to know anymore what type e is.

All it needs to do is to make a polymorphic call to get\_info, to get the proper employee information.

You just saw how switching on object types violates the open-closed principle. You also saw how to

solve the problem with polymorphism.

At this point, extending functionality becomes very easy.

Let me add a new employee type to the system. The employee type is a programmer. It inherits from

employee and adds an extra attribute called programming\_language.

Since we adhere to the open closed principle, all we need to do is to add a new class that overloads

the get\_info method properly.

And instead of having to add an extra case to check if the type of e is programmer, we did not have

to change to print\_employee function at all.

And that is what we call open for extension, closed for modification.

Another huge benefit of working with the open-closed principle is that removing functionality can be

done relatively safe.

Let's say that the company finally realized it does not really need managers and asks you to remove

everything that has something to do with manager objects.

You don't have to think about this too long. Since there are no switches in the code, that check for

the manager object type, you can just delete the manager class.

Perhaps you will have to clean up some places where managers were instantiated in the code, but that

is as easy as running the code and see what errors occur when the code is compiled.

And that's it, you just learned one of the most important principles in software engineering.

Whenever you see if-else statements or switches that select on object types, you might be looking at

a violation of the open-closed principle.

The open-closed principle is one of my favorites and is closely related to the next principle, the

Liskov substitution principle.

Chapter three. Liskov substitution principle.

This principle is all about expectations.

The idea behind this principle is that subclasses should not change the behavior of super classes in

unexpected ways.

This principle is closely related to the open-closed principle.

Violations of the Liskov substitution principle usually show us where the open-closed principle is

violated.

The technical description of the Liskov substitution principle goes something like this.

If B is a subtype of A, objects of type A may be replaced with objects of type B without breaking things.

Sounds great, but what does it mean?

I will show that by deliberately violate the principle and break the design of the code.

Before I show the code, I already want to tell you how to adhere to the principle. Again, we use

polymorphism.

You might wonder, isn't there anything polymorphism cannot do for us?

Well, the answer is yes.

Liskov substitution violations sometimes also indicate design flaws, and here polymorphism won't save

us.

We just have to redesign parts of the code.

I will show four ways to violate the Liskov substitution principle.

One. Selecting on types.

Two. Break the is-a relationship of inheritance.

Three. Raise errors in overridden methods.

Four. Breaking constraints.

Let's start with the first example of a violation of the Liskov substitution principle.

It is one of the most common violations of the principle. A selection on types. We have already seen

this problem in the last chapter on the open-closed principle.

Let me quickly repeat what it was about.

There is a base class.

And a subclass that implements extra attributes.

A function that takes a reference to an employee object now has to check what type of object e is.

In order to check for the type, the switch must know all possible employee types, and so it violates

the open-closed principle.

And since supertype employee cannot be replaced by subtype manager without checking the type of e,

the Liskov substitution principle is also violated.

You have seen how to solve the problem in the previous chapter.

The second example of a Liskov substitution violation is more subtle.

When classes inherit from each other, we say they have an is-a relationship.

For this example, I'll create an employee class and inherit an intern class from it.

We assume an intern is an employee.

I start with an employee class that stores a name and salary.

I create a method that prints the name and year salary.

Finally, I create an employee and call the method print\_year\_salary.

The system should also support interns. In our system, interns are employees without salaries.

Since the intern has no salary at all, the salary should not be zero, but set to None.

This is done in the initializer. The initializer is overridden and passes None to the super initializer

or.

That way, the salary will always be none, no matter what salary the intern is instantiated with.

This leads to a problem.

When the year salary for an intern is printed, an error occurs because although it looks like Dave's (edit: Chuck in the code)

salary is zero, the intern class overwrites to value with None.

Here is the full code.

The intern class changes the behavior of the superclass in unexpected ways by setting the salary to

none.

Perhaps you think, why not set the value to zero?

That way the code will not crash.

That's true, but that would create another problem.

For example, interns would lower the average salary of all employees, even if they don't have a salary

at all.

The solution here is simple, even if an intern is an employee, for all normal intents and purposes,

an intern object is not an employee object because the behavior is not consistent.

The print\_year\_salary function would have to check the employee type to work properly, but that would

violate the open-closed principle.

This is an example where polymorphism cannot help us, and the conclusion is that the intern class should

not be inherited from the employee class. An intern object does not have an is-a relationship with

an employee object.

You just saw that breaking the is-a relationship can violate the Liskov substitution principle. There

is another example that breaks

the is-a relationship.

It is caused when overridden methods raise errors that are not inherited from errors raised by the parent

class.

A common example is a not implemented error.

An employee can be promoted by calling the promote method.

Class intern inherits from employee, but an intern cannot be promoted. The promotion effort is overridden

and raises NotImplementedError.

The promote\_employee function takes an employee and calls promote on it.

The same problem as in the previous example occurs. An intern object is not an employee object because

the behavior is not consistent.

The promote\_employee function would have to check the type of e to know if it should call the promote

method.

The overridden promote method raises an error that is never raised by the method it overrides.

This is a violation of the Liskov substitution principle.

The last violation I'm going to show is ignoring class constraints. An example of this violation is

when subclasses change values without checking constraints.

Class employee takes an employee\_id and name. Both parameters are stored in the object.

The employee class has a method to check if the employee\_id is valid. The employee\_id must be a number

greater than zero.

Class Intern is added to the system and overrides the initializer. For now, it just calls the initializer

of the base class.

An Intern object is created and the system checks if the object is valid. The result is true.

Now, the personnel department wants to see a list of employees and asks if the employee IDs of the

interns can be prefixed with the letter I.

This would ask for a change in the reporting module, but time is short and the developer creates a hack.

The developer prefixes the letter I when the initializer of the superclass is called.

This will result in the desired report, but it also violates the employee ID constraint.

The constraint specified that the employee\_id must be a valid number, greater than zero.

When the is\_employee\_id\_valid method is called, it will return false.

The subclass has violated the constraint.

Conclusion.

It doesn't matter how tiny the change of behavior in a subclass is, as soon as the subclass starts

to change the behavior in unexpected ways, the subclass does not have an is-a relationship with its

superclass anymore. From our examples with the intern subclass,

there is a simple conclusion to draw.

An intern object is not an employee object, according to our system.

Liskov substitution principle violations can be difficult to detect and even harder to solve when the

system is already in use.

If we do not spot them early on, at one point, the only practical solution is to implement if-else

statements and thus break to open-closed principle. Sometimes it would be better just to accept right

from the start that some classes do not have an is-a relationship.

Chapter four. Interface Segregation Principle.

This chapter is all about keeping interfaces cohesive.

You know what the purpose of an interface is, an interface defines a signature that must be implemented

by classes that implement that interface.

That same interface can now be used as a parameter type for constructors or properties of other classes.

This is the magic behind dependancy inversion that we will look at in the next chapter.

For now, we will focus on the interface itself.

I hear you think. Interface? But Python does not have interfaces.

That's true.

We can create base classes or abstract base classes and use multiple inheritance to inherit from them.

But in a dynamic language like Python, an interface does not have to be a physical thing at all.

An interface exists at the moment when we specify it in our designs.

For example, look at the class diagrams on the screen. Both classes have a method raise\_salary

that seems public.

Can you guess what the common interface for both classes is?

Here it is. The task of an interface is to describe the signature of the class.

Now we know what an interface is. So what is the interface segregation principle?

The interface segregation principle says that no client should be forced to depend on methods it does

not use. This sounds plausible, and yet it is very easy to violate the principle.

Let me give you an example of such violation.

We are going to write a class for an iPhone, our iPhone can make phone calls and can be unlocked by

swiping.

Here is the code.

You can see the method to make a phone call and a method to unlock the phone. If we would extract an

interface from this class, it would look something like this.

At this point, you could create a bunch of smart phones that all implement the phone interface.

But now we want to create a class for a Nokia 2720.

This is a feature phone without a touchscreen.

When we implement the code for this phone, we see a problem.

What does this phone do when swipe\_to\_unlock is called?

Right now, it would raise an error and the problem is that there is not much we can do to improve the

situation.

The best thing we can do is override the method and add some information to the error, but the reality

is that this method should not be known at all in the Nokia class.

Clients of the class would always have to check if the phone is a Nokia to prevent calling swipe\_to\_unlock

.

As you have seen in Chapter three, this violates the Liskov substitution principle, but why does

it also violate the interface segregation principle?

The phone interface has two roles, one role has something to do with making phone calls and the other

has something to do with touch screens, but both roles are combined in a single interface.

The interface forces a phone without a touch screen to deal with touch functionality.

That interface is not coherent.

This is a violation of the interface segregation principle.

The solution is to break up the interface by its roles.

Here is the old interface called Phone.

I'm going to split it in two interfaces.

One interface for phone calls and one for touch functionality.

Now, the iPhone class can use multiple inheritance to inherit from both interfaces.

And the Nokia class only inherits from the phonecall interface. The swipe\_to\_unlock method is now fully

unknown to the Nokia class.

We have successfully broken up the single incohesive interface into two cohesive interfaces that are grouped

by their rules and do not violate the interface segregation principle anymore.

So what happens when new functions are added to the phone classes?

Let's start with an emergency call.

Before we started with interface segregation, we would have asked ourselves this question: what phones

can make an emergency call?

But now we ask ourselves the question to what role belongs making phone calls?

The answer is easy.

An emergency call should be added to the phonecall interface.

Here is another example, text messages.

All phones support text messages, so in what interface do we put the method?

We could put it into phonecall interface, but that would add another role to the phonecall interface

and violates the interface segregation principle.

The solution is clear, we need to introduce a third interface.

And this is how interface segregation works, we break up the interface in groups of functions.

This prevents subclasses from being polluted by methods they do not use.

Conclusion.

Whenever clients depend on so-called 'fat interfaces', but only use few methods from it, you might be

looking at a violation of the interface segregation principle.

The goal is to create cohesive interfaces by breaking up in groups of functions.

Watch out that breaking up interfaces does not become the goal itself.

It is possible to overdo interface segregation.

Finding a good balance is key here.

You have seen the first four principles of solid.

We started with the single responsibility principle that states that things should only have one reason

to change.

We continued with the open-closed principle while we saw it is indeed possible to extend software with

a minimal amount of changes in existing code.

The Liskov substitution principle showed how important it is that subclasses do not change the behavior

of super classes in unexpected ways.

And we just have seen how to break up interfaces to adhere to the interface segregation principle.

If I would have to pick two favorite principles, it would be the open-closed principle and the one

we are going to look at next.

A dependency inversion principle.

What is it about? You'll learn that in the next and final chapter of this course.

Chapter five. Dependency inversion principle.

This is the final topic of this course, it provides the mechanism behind most other principles.

What is its definition?

High level modules should not depend on low level modules.

Instead, they should depend on abstractions.

This can be difficult to grasp.

Let me show you what it means.

Your program starts in main.

Main calls function f in module M.

M calls function g in module N.

I have a question for you. Is this an object oriented design?

The answer is we don't know.

It could be anything, it could be a procedural design.

It could be OO. All we know is that high level modules call functions in lower level modules.

The higher level modules depend on the lower level modules.

In order for main to use module M, main needs to import M.

In order for module M to use module N, M needs to import N.

We call this a source code dependancy.

High level modules know about the lower level modules. They target concrete classes.

This design comes with two problems.

Concrete classes are volatile.

Depending on concrete classes breaks the open-close principle.

What does the first problem mean?

Concrete things change a lot.

Abstract things change much less frequently.

If high level modules depend on low level modules, changes in low level modules trigger a recompile

in all higher level modules.

Historically, this was a problem because of compile times. In 2021

this is not a big problem anymore.

The second problem is more serious.

So where do things start to go wrong?

Imagine the software for a cash register. Each time a purchase is made, a receipt is printed.

I create a class called Reporting. The class contains a method called print\_receipt that takes a string

with the receipt text.

At this point, we need to send the text to a printer.

We will use the internal cash register printer for this. Let's create a class for it.

A module printers.py is created and has a class CashRegisterPrinter.

How do we use the class?

The reporting module needs to import from the printers module.

Then it instantiates a new CashRegisterPrinter. This code will work, but we have created a dependency

on a concrete class.

This is a violation of the dependency inversion principle.

And by doing so, it also violates the open-closed principle.

Let me demonstrate this by adding a new printer.

Instead of printing to the internal cash register printer, a laser printer will be used.

Let's create a class for the laser printer in printers.py.

printers.py now has two printer classes.

One for the internal cash register printer and one for the laser printer. We have extended the software

but need to modify existing code to use it.

In order to change the printer, the reporting module needs to be changed.

This is a violation of the open-closed principle.

This time it is caused because we depend on concrete classes instead of on abstractions.

So how do we solve the problem?

Let me show a dependency diagram of the current code.

Main imports Reporting.

Reporting imports LaserPrinter.

Now you are going to see what inversion means in dependency inversion.

The first thing we do is to introduce a printer interface. This interface will just exist in our design.

In a dynamic language like Python, we do not physically have to implement interfaces to get polymorphism.

LaserPrinter inherits the printer interface.

And the reporting module depends on the printer interface.

All arrows used to point downwards. Modules depended on lower level modules.

But now Reporting depends on an interface, it depends on something abstract.

LaserPrinter depends on that same abstraction.

This is a break in the dependency chain.

It does not matter what happens in LaserPrinter. The reporting module doesn't even know there is such

a thing as a laser printer.

All it knows is that there is something with a method called print\_receipt.

So how do we hook this up in Python?

Let's start from the top and implement main.

The first thing you notice is that all the dependencies are imported here.

A LaserPrinter object is instantiated.

And then a Reporting object. And this is the place where the printer needs to be injected.

To do so, the reporting class gets a class initializer that allows for the printer object to be injected.

Let me show the code.

A dunder init method is added and the printer argument is stored in self. self.printer now holds

an instance of a printer object.

This printer object can now be used to print the receipt.

Notice there are no imports and no object instantiation is happening here.

There is no dependency on a concrete class.

We can now pass the printer object to the reporting class.

Finally, print\_receipt is called on the reporting object. We have now solved the dependency inversion

principle violation.

And this is the point where we start to benefit when the requirements change.

For example, what happens if we switch back to the internal cash register printer?

We swap the laser printer for a cash register printer in main.

The reporting class does not need to be modified anymore to support another printer.

Perhaps you are wondering what we have gained here. We still have to change code to support a new printer.

That's true.

The open-closed principle does not prevent code modification, but it confines it to specific places.

Main is an excellent place to create the dependency tree that will be used in the rest of the program.

If you are worried about an explosion of import statements in main, there are techniques to help preventing

that.

You can use factories.

A factory is an object that creates new objects. That way you can move the logic out of main and into

a factory.

If you are interested in this, you could watch my course on the Python object model where I show an

example of a factory.

Some people like to use IOC container frameworks to manage the dependencies for them. The dependency

tree is configured in a config file and resolved by the framework when needed.

Personally, I favor writing out the dependency tree in source code. This gives me full control over

the life span of each and every object in the code.

As always, there is a solution for each scenario and you have to find out what works best for you.

Conclusion. The dependency inversion principle is the mechanism behind many of the other principles.

Especially the open-closed principle.

It allows us to depend on interfaces rather than upon concrete classes.

The symptoms of violations of the dependency inversion principle are

Object instantiation in lower level modules and the import statements needed for these instantiations.

These violations can be solved by injecting dependencies.

And that concludes the last chapter of this course. You have now seen the five S.O.L.I.D. principles.

Now let's look at the principles as a whole.

What are the main lessons to be learned?