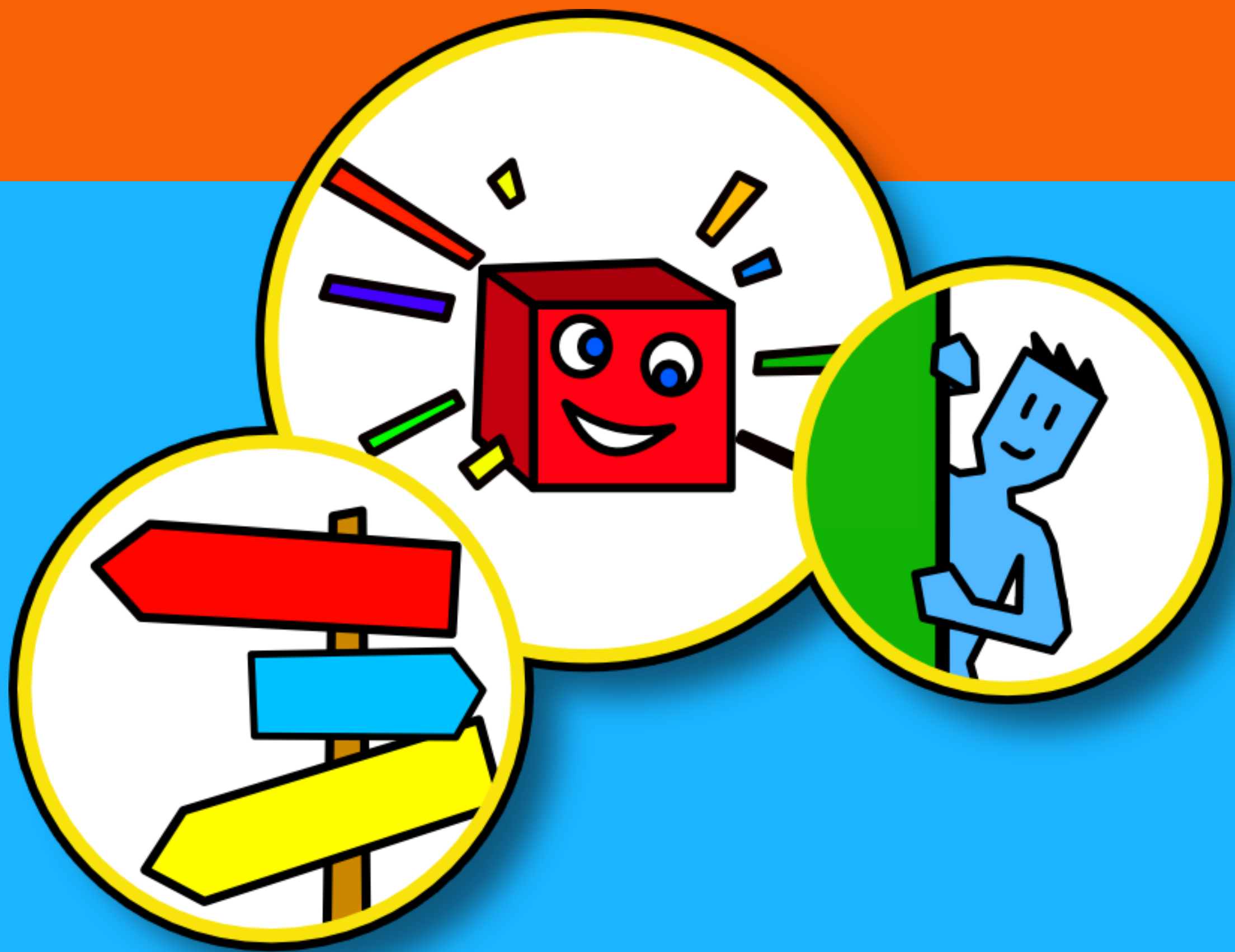


THREE COMMON SOFTWARE ENGINEERING PROBLEMS AND SOLUTIONS



Loek van den Ouweland
Software Engineering



THREE COMMON SOFTWARE ENGINEERING PROBLEMS AND SOLUTIONS

eBook version 1.0

©2024 Loek van den Ouweland

Illustrations: Stephanie Wunsch

www.pythonforeveryone.com

Software Engineering:

3 common problems and solutions

When I was a kid, I loved playing with Lego and Playmobil, listening to records and reading comics. Preferably all at the same time!



Every now and then, my mother would ask me to tidy up the mess so she could clean my room. But there was a problem. I did not see the "mess". Only when I grew up and had to clean the room myself, I saw the problem. It is annoying to start cleaning without first tidy-ing up the place!

But when you don't have to clean yourself, tidy-ing up things seem like a **solution without a problem!**

Beginner software developers experience something similar in their work. More experienced colleagues will tell them they need pattern A or principle B. But why? Why do they even need a solution when they don't recognize there is a problem?

This haunted me for years during my career. I have made countless games and apps for Windows, macOS, and the web, and many more things that required me to write code.

And although 98% of all the code I wrote in my life is now gone, I have made many projects that have been used for years by my customers.

But I'll tell you a secret. The code I wrote worked, but were no marbles of software engineering! I knew how to use the tools and languages to achieve my goals, but I always regretted not diving deeper into common computer science patterns.

I admired colleagues who knew problem X could be solved with solution Y. And envied colleagues who even anticipated challenges and knew how to address them before becoming a problem!

But unfortunately, the best developers are usually not the best educators. They can show you the problem but without explaining **why** it is a problem, you will fail to recognize it in future code.

You and I are going to do things differently!

In this eBook, you are going to learn 3 common problems, how to recognize them in your code and how to solve them. They are:

1. Switching on types. Adhere to the open-closed principle with polymorphism
2. Objects that create objects. Solve coupling with dependency injection
3. Dependencies on libraries. Hide them behind an interface

Problem 1

Switching on types



**Adhere to the open-closed
principle with
polymorphism**

The open-closed principle says:

"Software entities should be open for extension, but closed for modification."

Software entities can be anything from modules, classes to functions. In this example I will work with functions. Let me rephrase the open-closed principle in my own words:

*"I want to change the behavior of a function without changing the code **in** that function."*

That sounds like a contradiction, right? It is and yet I will show you how to achieve this! But before I can, I have to show you the problem we are going to solve.

I will do this with an example where the open-closed principle is violated and what problems that causes. Then I will fix the problem. After you have seen the problem and solution, you will be able to spot the problem in your own code and know how to solve it.

See it online



```

class Header:
    def __init__(self, text: str) -> None:
        self.text = text

class LiteralText:
    def __init__(self, text: str) -> None:
        self.text = text

document = [
    Header("The open-closed principle"),
    LiteralText("Open for extension, closed for modification.")
]

def show_document(doc):
    for element in doc:
        if type(element) is Header:
            print(element.text.upper())
        elif type(element) is LiteralText:
            print(element.text)

show_document(document)

```

main.py

Classes "Header" and "LiteralText" take and store text. Although the classes are very similar, both need to exist in order to switch on their types in the "show_document" function and print header text all-caps, and literal text as-is.

A switch like this is a problem because every time new document element classes are added to the system, the switch also needs to be changed.

The "show_document" function cannot be closed (locked) and with each new class to switch on, there is the risk to break something.

Perhaps you have noticed I did not annotate 100% of the code. For example, I omitted the "doc" parameter annotation in the "show_document" function. In order to annotate it, I would have to create a common interface like a base class or protocol class. Since static type checkers allow gradual typing, I was able to omit that class to keep the code small.

The problem

```
class Header:
    def __init__(self, text: str) -> None:
        self.text = text

class LiteralText:
    def __init__(self, text: str) -> None:
        self.text = text

class HorizontalLine:
    def __init__(self, width: int) -> None:
        self.width = width

document = [
    Header("The open-closed principle"),
    HorizontalLine(25),
    LiteralText("Open for extension, closed for modification.")
]

def show_document(doc):
    for element in doc:
        if type(element) is Header:
            print(element.text.upper())
        elif type(element) is LiteralText:
            print(element.text)
        elif type(element) is HorizontalLine:
            print("=" * element.width)

show_document(document)
```

main.py

Class "HorizontalLine" is added to the code and the "show_document" function also needed to be changed. This is a violation of the open-closed principle. The function should be closed for modification, while allowing extending its behavior.

Whenever you hear yourself say: For type A, do this and for type B do that and for type C do that, you are possibly violating the open-closed principle. And how can we solve it? With polymorphism!


```
class Header:
    def __init__(self, text: str) -> None:
        self.text = text

    def get_representation(self) -> str:
        return self.text.upper()

class LiteralText:
    def __init__(self, text: str) -> None:
        self.text = text

    def get_representation(self) -> str:
        return self.text

class HorizontalLine:
    def __init__(self, width: int) -> None:
        self.width = width

    def get_representation(self) -> str:
        return "=" * self.width

document = [
    Header("The open-closed principle"),
    HorizontalLine(25),
    LiteralText("Open for extension, closed for modification."),
]

def show_document(doc):
    for element in doc:
        print(element.get_representation())

show_document(document)
```

Each class gets method "get_representation" that returns the appropriate text representation. The "show_document" function can be simplified drastically. Instead of switching on types, it can call method "get_representation" on each document element.

This code adheres to the open-closed principle. The "show_document" function is now closed and does not need to change even if new document elements are added.

The reward

```
class Header:
    def __init__(self, text: str) -> None:
        self.text = text

    def get_representation(self) -> str:
        return self.text.upper()

class LiteralText:
    def __init__(self, text: str) -> None:
        self.text = text

    def get_representation(self) -> str:
        return self.text

class HorizontalLine:
    def __init__(self, width: int) -> None:
        self.width = width

    def get_representation(self) -> str:
        return "=" * self.width

class BulletList:
    def __init__(self, bullets: list[str]) -> None:
        self.bullets = bullets

    def get_representation(self) -> str:
        return "\n".join("- " + x for x in self.bullets)

document = [
    Header("The open-closed principle"),
    HorizontalLine(25),
    LiteralText("Open for extension, closed for modification."),
    BulletList(["Open", "Closed"])
]

def show_document(doc):
    for element in doc:
        print(element.get_representation())

show_document(document)
```

main.py

Class "BulletList" was added to the code and an instance was packed in the document list. Notice that the "show_document" function will now produce a different result without modifying it.

That is the open-closed principle at work. And it works the other way around too. When document element classes are removed from the system, they only need to be removed from the document list and the "show_document" function will keep working without changes.

All good news?

So is it all good news? Nope. Just like you can over-eat, you can also over-refactor. Instead of a single, local switch, the code is now scattered over the project.

My advice is that you do **not** start with thinking about patterns and principles but to just start with your project and once you discover problems, start refactoring. In other words, do not search for problems where there might be none.

The solution I showed is very elegant but to understand the code, you need to look at more than one place. When related code is kept together, we speak of Locality of Behavior. The updated code, while very elegant, moved away from that principle.

"When related code is kept together, we speak of Locality of Behavior. The updated code, while very elegant, moved away from it a bit."

What I wished I understood earlier

I always found it a bit misleading to say "**closed for modification**". What I wished I knew earlier about the open-close principle, is that closing does not mean locking a complete class or module. Instead, it aims to close down much smaller units, like a single function.

If you look closely, all we did was move the text representation logic to the respective classes in polymorphic method "get_representation". That is what allows us to close the "show_document" function and let the classes handle the text representation. All the function needs to do, is call "get_representation" on each object and Duck Typing is used to check for the presence of the method. If the method is found, it is called. If not, a type error is raised by Python.

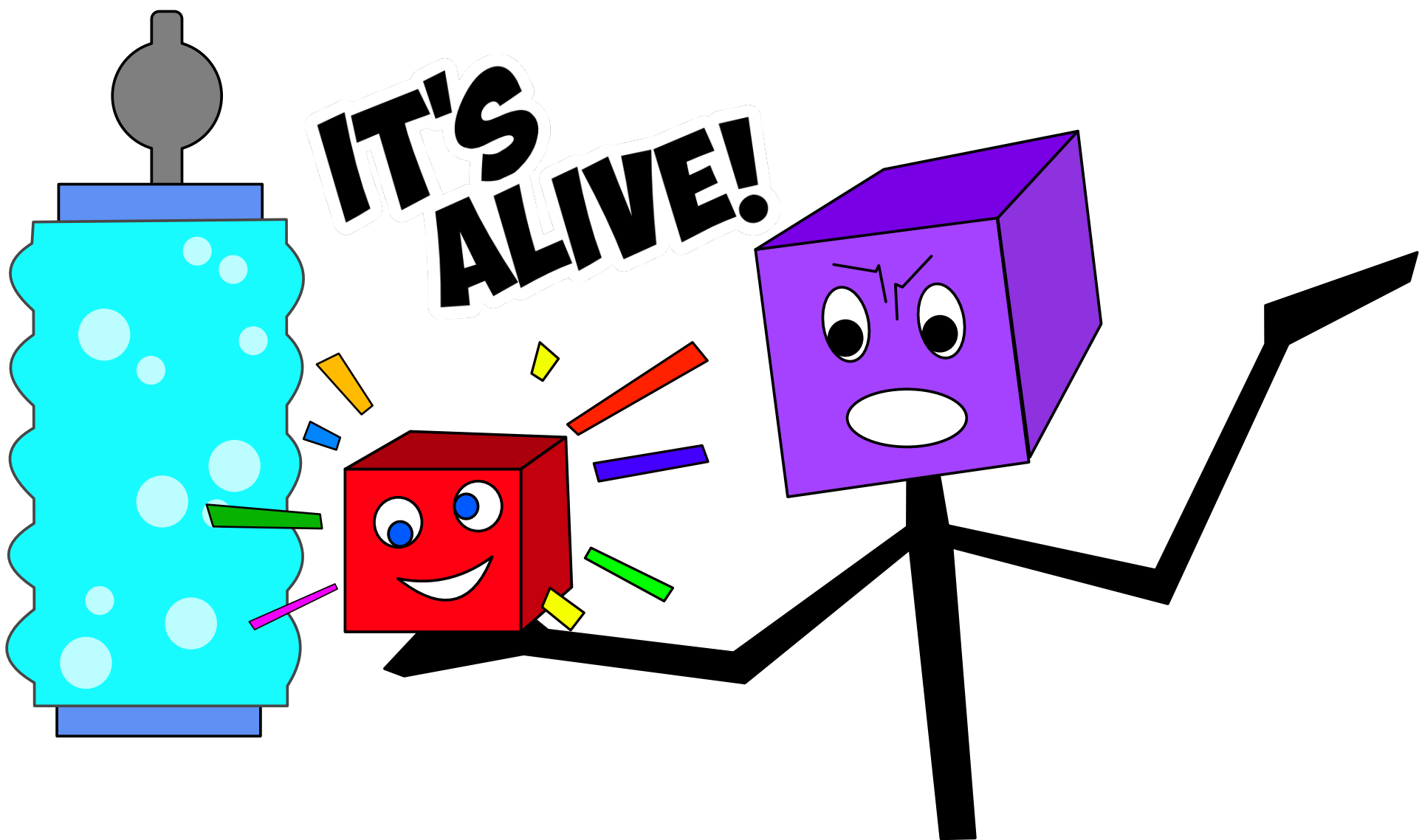
Problem 2

**Objects that create
other objects**



**Solve coupling with
dependency injection**

In this chapter, you will see a class that creates objects. You learn how this couples code and why that is a problem.



Dependency Injection is a technique where functions and objects receive objects that are required to do a job. An example would be a function that loads data and then processes it. Instead of the function loading the data directly, it receives an object that can perform the data loading. By doing so, the function has at least one less responsibility.

When responsibilities are spread over different classes, we speak of "separation of concerns". The fewer responsibilities a class has, the less reasons it has to change. The less change, the less risk!

But these concepts are very abstract and will not help you much at this point. In order to get it, you first must understand the problem and for this, I will show you a real-world example.

In 2013, me and my team created the Windows version for Wunderlist in Berlin.



At that time, Microsoft was working on a unified Windows 10 platform but Windows 8 and Windows Phone 8 were still based on different Windows Operating System versions. Although 80% of the code could be re-used, there was a big chunk of code that was platform dependent.

For example, loading and saving data worked differently across platforms.

Although back then our code was written in C# and XAML, I will use Python to demonstrate the problem.

```

class Data:
    def __init__(self, platform: str) -> None:
        self.platform = platform

    def load(self) -> None:
        if self.platform == "Windows8":
            print("Loading from Windows 8.")
        elif self.platform == "WindowsPhone8":
            print("Loading from Windows Phone 8.")
        print("Deserializing data.")
        self.data = {}

    def save(self) -> None:
        print("Serializing data.")
        if self.platform == "Windows8":
            print("Saving to Windows 8.", "{ data: ... }")
        elif self.platform == "WindowsPhone8":
            print("Saving to Windows Phone 8.", "{ data: ... }")

```

data.py

Look at this code. The class initializer takes a platform string. The load method switches on it to load data from the appropriate platform. It then mixes this functionality with deserializing data.

The save method serializes data and then mixes this functionality with saving on the appropriate windows platform. "main" hooks everything up.

```

from data import Data

data = Data("Windows8")
data.load()
print("Do something with data")
data.save()

```

main.py

There is a problem with this code. Class "Data" violates the Single Responsibility Principle because it mixes loading and saving with processing data. Fixing the problem goes in two steps:

1. Extract the loading and saving functionality in individual classes and methods
2. Abstract and decouple

Step 1: Extract storage functionality

Module "storage" adds two classes to the system. All methods have a single responsibility which is the physical loading or saving of data.

```
class Windows8Storage:
    def load(self) -> str:
        print("Loading from Windows 8.")
        return "{ data: ... }"

    def save(self, data: str) -> None:
        print("Saving to Windows 8.", data)

class WindowsPhone8Storage:
    def load(self) -> str:
        print("Loading from Windows Phone 8.")
        return "{ data: ... }"

    def save(self, data: str) -> None:
        print("Saving to Windows Phone 8.", data)
```

storage.py

The next logical step is to create the proper storage instance in "Data", based on the platform. "main" remains unchanged.

```
from storage import Windows8Storage
from storage import WindowsPhone8Storage

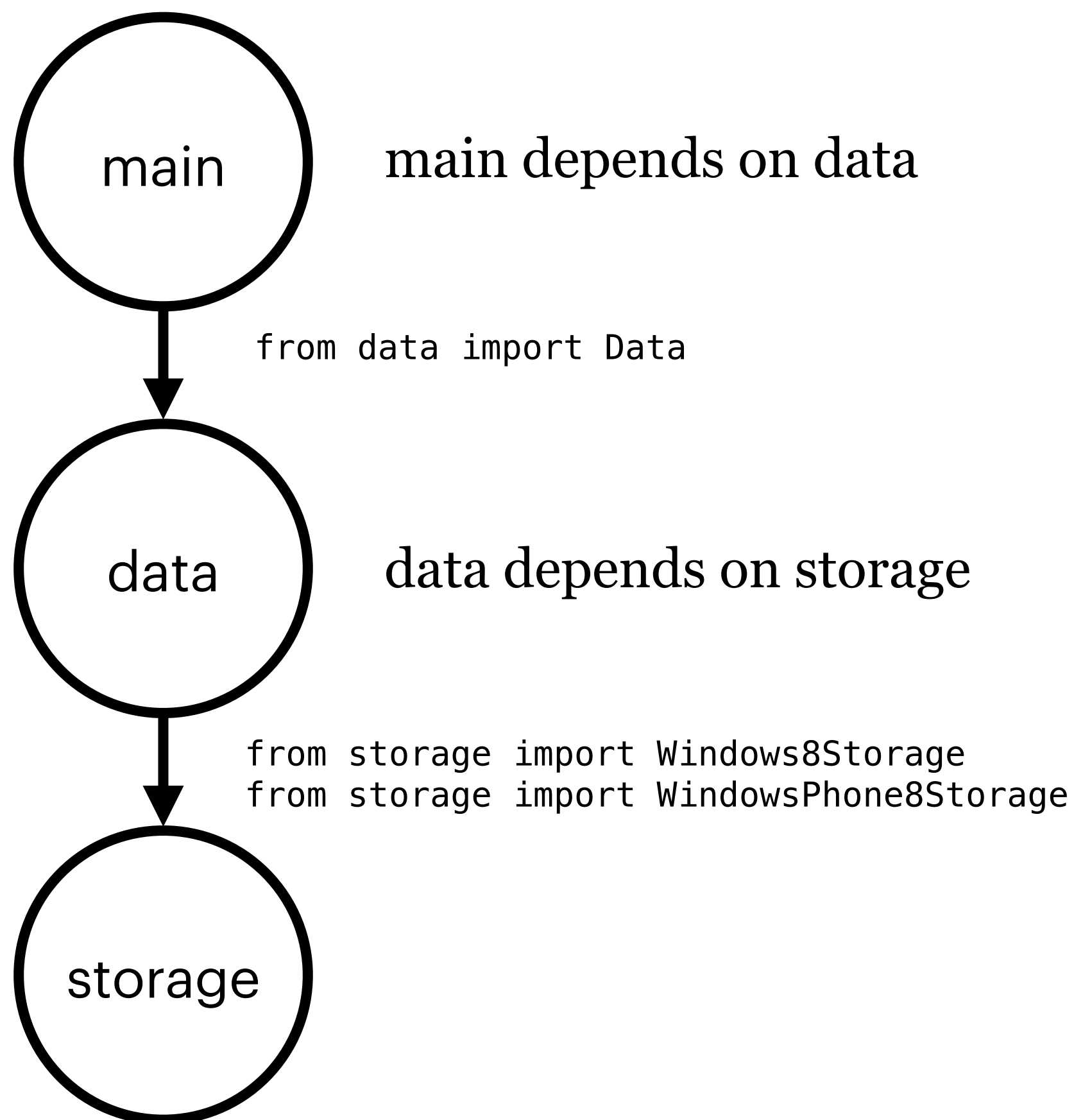
class Data:
    def __init__(self, platform) -> None:
        if platform == "Windows8":
            self.storage = Windows8Storage()
        elif platform == "WindowPhone8":
            self.storage = WindowsPhone8Storage()

    def load(self) -> None:
        self.storage.load()
        print("Deserializing data.")

    def save(self) -> None:
        print("Serializing data.")
        self.storage.save("{ data: ... }")
```

data.py

The system already has much improved. Each part of the code has a single responsibility. Here is the dependency diagram. It is created by following the import statements.



The main module may have knowledge of all other modules. But other modules must prevent "knowing" classes in other modules and in this example, to create the storage instances, data must have a source code dependency to storage. The data module is **coupled** to the storage module.

Even if the load and save method in "Data" are very clean, the class initializer creates instances of storage classes. The class still mixes object creation and other functionality.

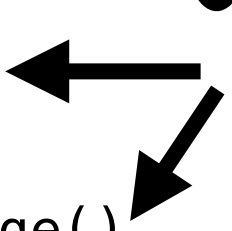
```
from storage import Windows8Storage
from storage import WindowsPhone8Storage

class Data:
    def __init__(self, platform) -> None:
        if platform == "Windows8":
            self.storage = Windows8Storage()
        elif platform == "WindowPhone8":
            self.storage = WindowsPhone8Storage()

    def load(self) -> None:
        self.storage.load()
        print("Deserializing data.")

    def save(self) -> None:
        print("Serializing data.")
        self.storage.save("{ data: ... }")
```

**COUPLED TO
STORAGE
CLASSES**



One of the problems with this, as you have already seen in chapter 1, is that switching on types violates the open-closed principle. Each time, storage classes are added or removed, the switch as well as the import statements need to be changed. In other words: existing, working code needs to be changed.

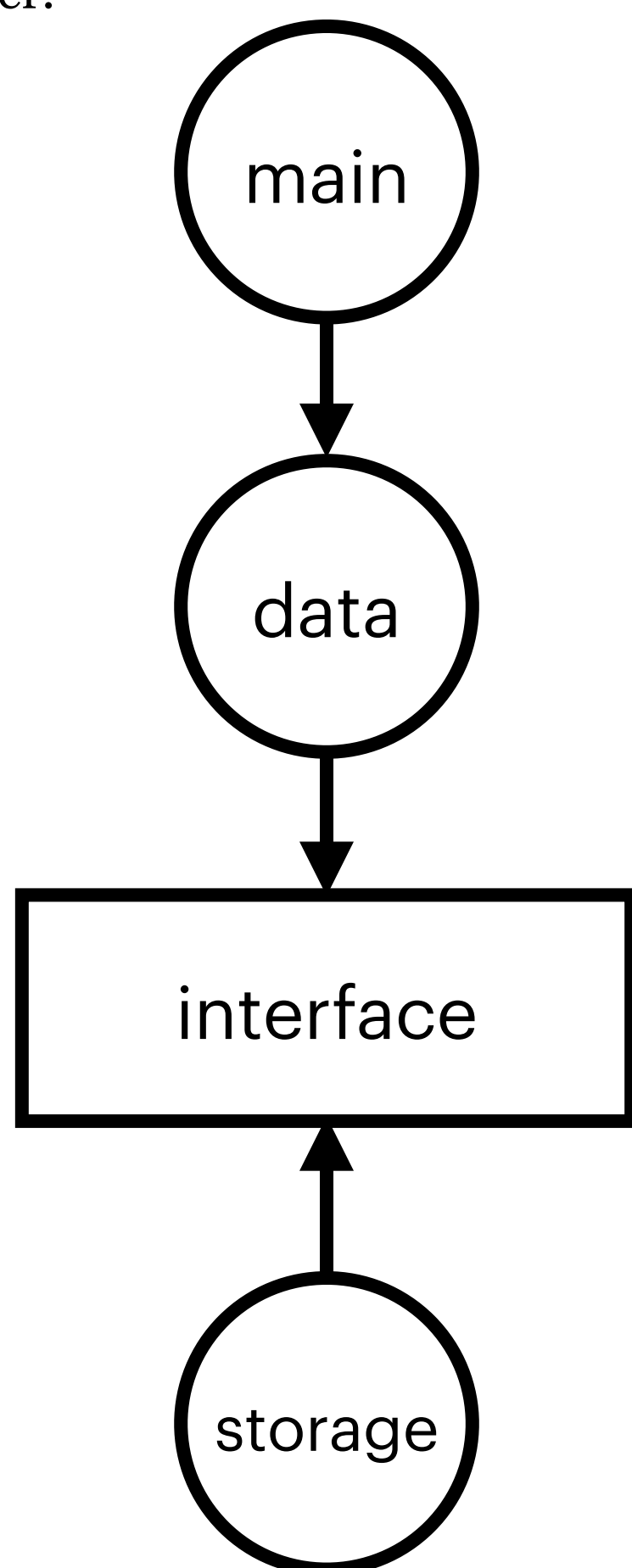
Aside from violating both the single responsibility principle and the open-closed principle, there is a technical problem that programmers in a statically typed language recognize immediately. When a higher level module has a source code dependency on a lower level module, a cascade of recompilation is triggered by the change in that lower level module. This increases the overall compile time and becomes noticeable in large projects.

To sum up the consequences:

- The open-closed principle is violated. When storage classes are added or removed, the data module must also change
- The single responsibility principle is still violated since the data module is responsible for creating the proper object and then using it
- In a compiled language, changing implementation details in the storage module requires re-compiling the data module

Step 2: Abstract and decouple

The solution is decoupling with the help of abstraction. And abstraction is achieved with interfaces. Python does not have interfaces but it does have protocol classes, which allow structural subtyping that can be type checked with a static type checker.



An interface is inserted between data and storage. Notice that both "data" and "storage" depend on the interface.

The dependency from "data" to "storage" is now inverted and this technique is called **Dependency Inversion**.

In the code, this will be achieved by combining a few techniques.

First, I create an interface, which will be a protocol class. The interface will be hosted in new module "interfaces"

```
from typing import Protocol
```

interfaces.py

```
class Storage(Protocol):
    def load(self) -> str: ...
    def save(self, data: str) -> None: ...
```

Instead of passing a string to the class initializer in "Data", a storage instance is passed. Notice the type is not a concrete storage class but the storage interface. Class "Data" is now decoupled from the storage module. All it knows, is there is a "load" and "save" method but it has no idea where the actual data is coming from.

```
from interfaces import Storage
```

data.py

```
class Data:
    def __init__(self, storage: Storage) -> None:
        self.storage = storage

    def load(self) -> None:
        self.data = self.storage.load()
        print("Deserializing data.")

    def save(self) -> None:
        print("Serializing data.")
        self.storage.save("{ data: ... }")
```

All that needs to happen now, is that "main" instantiates the proper storage instance and pass it when instantiating an instance of Data.

```
from data import Data
from storage import Windows8Storage
from storage import WindowsPhone8Storage
```

main.py

```
storage = Windows8Storage()
data = Data(storage)
data.load()
print("Do something with data")
data.save()
```

You might ask yourself how the interface is enforced by Python. The answer is that it is not enforced by Python but checked with a static type checker like MyPy or Pyright. Most modern code editors enable static type checking automatically once you create a Python script. You can now experiment by renaming the save method to "save2".

```
from typing import Protocol
```

interfaces.py

```
class Storage(Protocol):  
    def load(self) -> str: ...  
    def save2(self, data: str) -> None: ...
```

In Visual Studio Code, static type checker Pyright spots the problem in main and pops up this error:

```
from data import Data  
from storage import Windows8Storage  
from storage import WindowsPhone8Storage
```

main.py

```
storage = Windows8Storage()  
data = Data(storage)  
data.load()  
print("Do something with data")  
data.save()
```

"Windows8Storage" is incompatible with protocol "Storage". "save2" is not present

And in data.py

```
def save(self) -> None:  
    print("Serializing data.")  
    self.storage.save("{ data: ... }")
```

data.py

Cannot access attribute "save" for class "Storage"

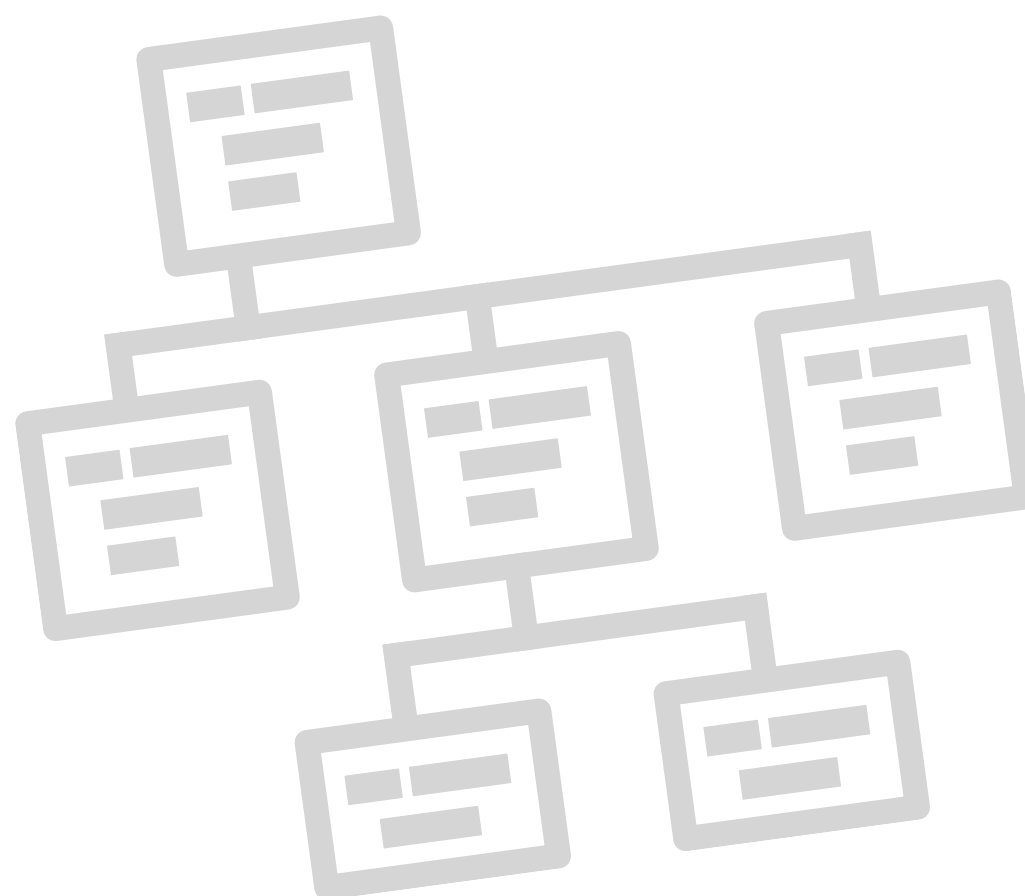
Structural subtyping

This kind of type compatibility is known as structural typing. I describe this and many more typing techniques in my course "The Python Type System" and if you want to check it out, try a few free chapters with the following link.



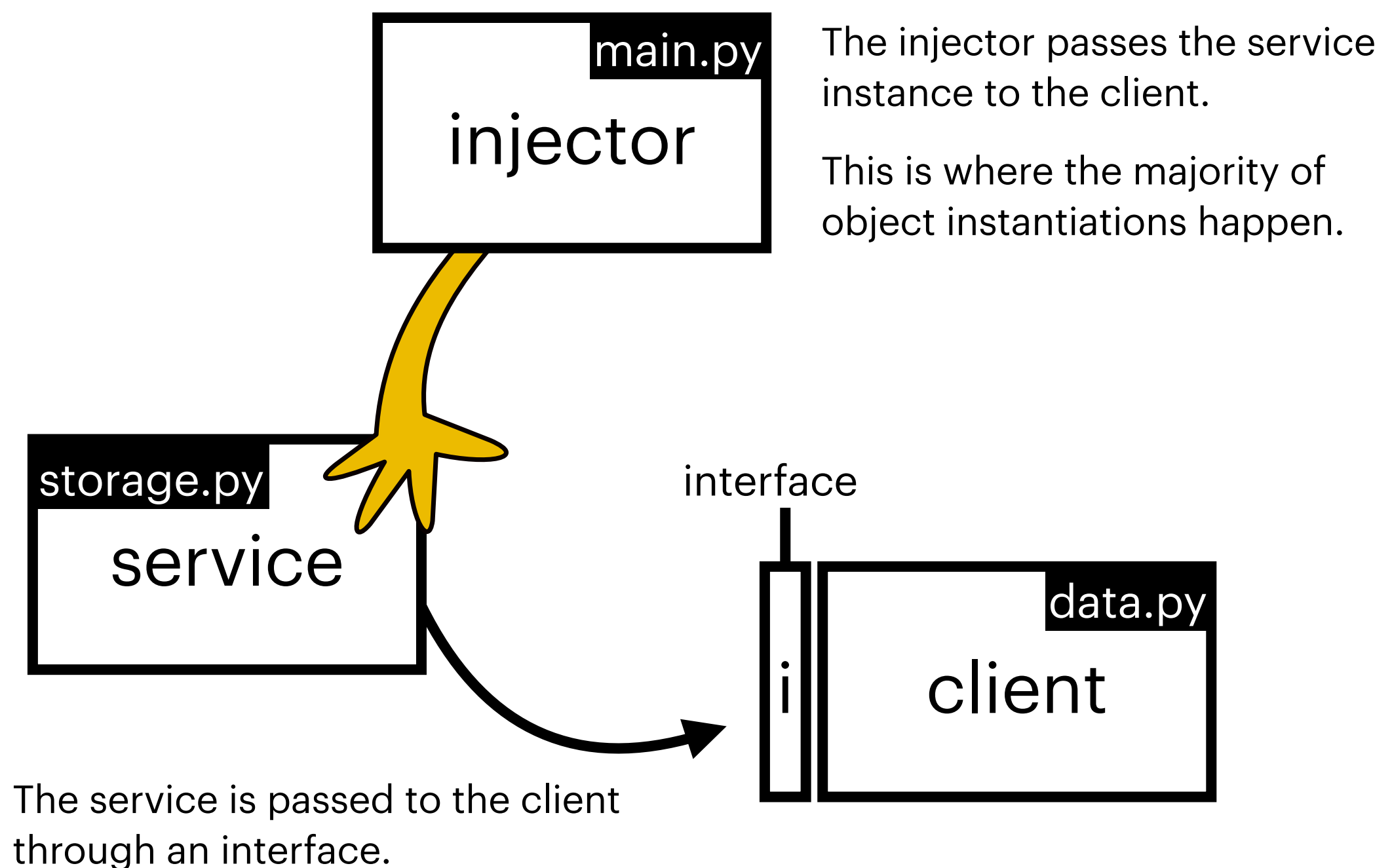
<https://pythonforeveryone.com/python-type-system-online-course.html>

Don't feel like typing? This QR will also bring you to the course:



The roles in dependency injection

The components in dependency injection have four roles: The injector instantiates service classes (or gets these instances from somewhere else) and passes them to clients. For example through the client class initializer. As long as the class initializer accepts abstract types (interfaces, abstract base classes), the service implementation can be changed without the client having to change as well. Since the client depends on an interface instead of the actual service type, the client is decoupled from the service.

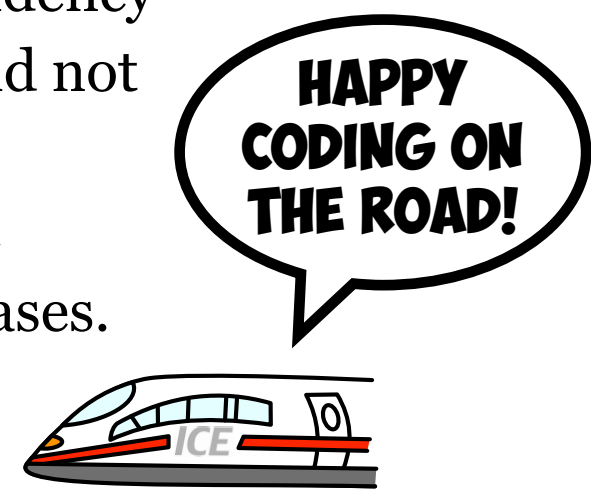


Statically typed vs. Dynamically typed

The interface is required in statically typed languages to get polymorphism. A dynamically typed language like Python does not technically require an interface. However, once you add type hints the static type checker will need an interface like a base class or protocol class (like in this example).

Benefits

One of the most obvious benefits of decoupling with dependency injection is unit testing. Before refactoring the code, it could not be tested without access to a physical storage device. The consequences of this are very serious. Such apps cannot be executed or tested without access to the networks or databases.



Our refactored code does things differently! A third storage class can be made that loads mock data:

```
class MockStorage:
    def load(self) -> str:
        print("Loading fake data.")
        return "{ data: { todos: [ ... ], lists: [ ... ] } }"

    def save(self, data: str) -> None:
        print("Do nothing.")
```

storage.py

The unit test can create a "MockStorage" instance and pass it to the "Data" class. Now all sorts of scenarios are possible including hard coded data during development and "real" data in production just by swapping storage instances at runtime.

All good news?

So is it all good news? Well, the system is decoupled and you can freely swap storage classes and even inject mock loaders for testing. But instantiating a concrete storage class is still hard-coded in main. You now have to use compiler directives to load the proper one or make a factory that does the instantiation for you. The decoupling sacrificed some Locality of Behavior.

And here is a lesson I had to learn at one point: you will always need some mechanism to instantiate objects. Dependency Injection does not relieve you from it. But it does allow you to concentrate object instantiation in few places instead of all over the project.

C, Java or C# programmers can check where they instantiate objects easily by running a global search for the "new" keyword.

Problem 3

Dependencies on libraries

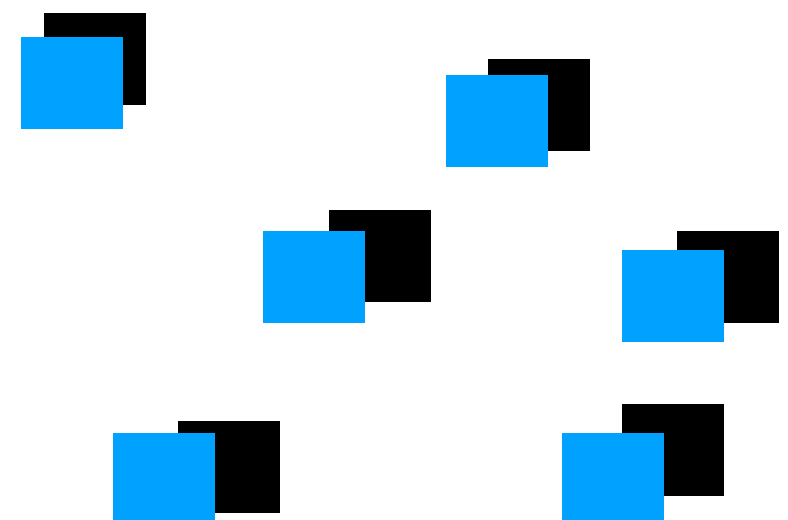


**Hide them behind an
interface**

We all love libraries! But oh, no! I have coupled my code to third-party code! Coupled code is problematic as you have seen in the previous chapter. But coupling your code to code that is not under your control is even worse!

A colleague of mine used to say: **Own your own problems**. He meant that with every library or framework, you import risk. You should always carefully weigh the benefits of ready-to-use third party code against writing the code yourself. But let's assume there is library X that does exactly what you need and you use it in your code.

If at one point you discover it has a problem, you should be able to swap it with library Y. But if library X sneaked into your code in many places, this can be a tricky exercise!



Hide the library

The solution is to hide the library behind an interface. But before I do, let me demonstrate the problem.

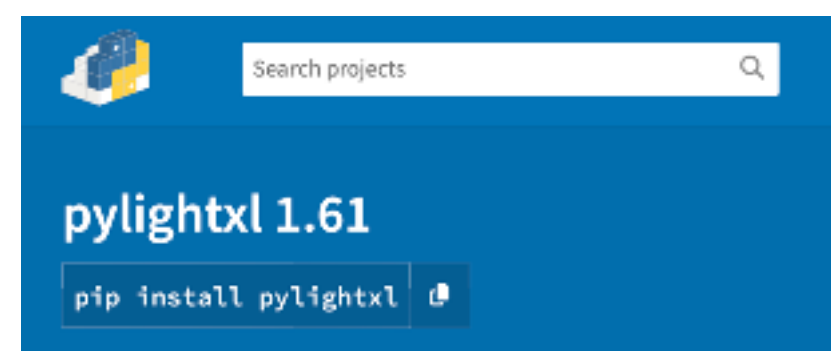
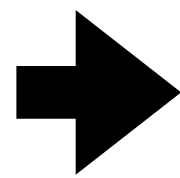
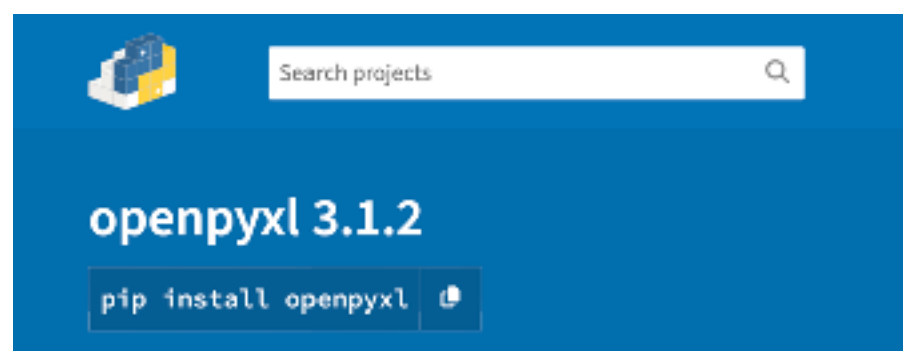
In this example, I will use a library to load excel files with Python. I will then show how hard it is to replace the library with another one, because the library code sneaked into our code in undesired ways.

employees.xlsx

Vera	2.000,00 €
Chuck	1.800,00 €
Samantha	1.800,00 €
Roberto	2.100,00 €
Dave	2.200,00 €
Tina	2.300,00 €
Ringo	1.900,00 €

Here is an excel sheet that is saved as "employees.xlsx". Python does not support reading excel files out-of-the box and needs a library for this.

I will start with a library called "openpyxl".



Since I will later replace it with another library called "pylightxl", I already pip-install both libraries:

```
pip install openpyxl  
pip install pylightxl
```

Here is the code to load and display the data from "employees.xlsx". Function "show_salaries" displays the name, salary and an increased percentage.

```
import openpyxl

def show_salaries(filename, sheetname, percentage: float) -> None:
    workbook = openpyxl.load_workbook(filename=filename)
    worksheet = workbook[sheetname]
    for name, salary in worksheet:
        print(name.value, salary.value, salary.value * (percentage + 100) / 100)

show_salaries("employees.xlsx", "Employees", 15)
```

main.py

Although the code is still small, there are problems with it.

- 1) The "show_salaries" function mixes the responsibilities of loading data and calculating the increased salary.
- 2) Proprietary library datatypes like worksheet cells have sneaked into our code.

I know in this example it looks harmless but once the loading functionality or business rules gets more complex, the risk of breaking things will grow. Let me show you what needs to happen to use the second library "pylightxl".

```
import openpyxl
import pylightxl

def show_salaries(filename, sheetname, percentage: float) -> None:
    # workbook = openpyxl.load_workbook(filename=filename)
    # worksheet = workbook[sheetname]
    # for name, salary in worksheet:
    #     print(name.value, salary.value, salary.value * (percentage + 100) / 100)

    workbook = pylightxl.readxl(fn=filename)
    worksheet = workbook.ws(ws=sheetname)
    for name, salary in worksheet.rows:
        print(name, salary, salary * (percentage + 100) / 100)

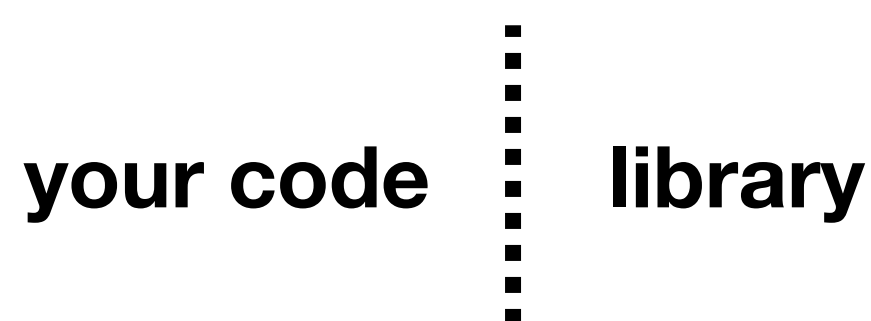
show_salaries("employees.xlsx", "Employees", 15)
```

main.py

In fact, pretty much all the code in "show_salaries" needed to be changed because the second library has a different API. Applying these changes, require you to test if the percentage calculation still works correctly.

But how? You can only do it manually because a unit test cannot test this code without access to an excel sheet. This is coupled code, where part of the code is not even under our control.

The goal of this chapter is to abstract away the library and have a clear boundary between excel stuff and our business logic.



You already guessed we need an interface for this. But what should it describe? Notice in the code that "openpyxl" needed to access cells with the "value" attribute. And "pylightxl" iterated the worksheet by accessing the row attribute. Their interfaces are different and contain proprietary datatypes.

The goal of this exercise is that all datatypes in "your code" are plain Python datatypes. All proprietary datatypes need to stay in the library. So we need conversion.

Make interface as simple and independent as possible

Separating our code from the library goes in two steps:

- 1) Let the interface use primary datatypes like lists, tuples, floats and strings
- 2) Make the data loaders convert their proprietary datatypes to primary datatypes

Here is interface "Excel".

```
from typing import Protocol
class Excel(Protocol):
    def load_employees(self) -> list[tuple[str, float]]: ...
```

interfaces.py

Classes "OpenPyXl" and "PyLightXl" are added to the system. Notice they both implement method "load_employees" with the same signature as in the interface. As you have seen in the previous chapter, this makes both classes structural subtypes of protocol class "Excel".

```
import openpyxl
import pylightxl

class OpenPyXl:
    def __init__(self, filename):
        self.filename = filename

    def load_employees(self) -> list[tuple[str, float]]:
        workbook = openpyxl.load_workbook(filename=self.filename)
        worksheet = workbook["Employees"]
        return [(str(r[0].value), float(str(r[1].value))) for r in worksheet.rows]

class PyLightXl:
    def __init__(self, filename):
        self.filename = filename

    def load_employees(self) -> list[tuple[str, float]]:
        workbook = pylightxl.readxl(fn=self.filename)
        worksheet = workbook.ws(ws="Employees")
        return [(str(r[0]), float(r[1])) for r in worksheet.rows]
```

plugins.py

Notice that both classes have very different implementations but both return the same datatype. A list of tuples where each tuple contains a string and float. Proprietary excel information is now completely hidden behind an interface with Python Standard Library Datatypes.

Main hooks everything up:

```
from plugins import OpenPyXl
from plugins import PyLightXl
from interfaces import Excel

plugin = OpenPyXl("employees.xlsx")
#plugin = PyLightXl("employees.xlsx")

def show_salaries(excel: Excel, percentage: float) -> None:
    data = excel.load_employees()
    for name, salary in data:
        print(name, salary, salary * (percentage + 100) / 100)

show_salaries(plugin, 15)
```

main.py

Function "show_salaries" is decoupled from concrete Excel libraries. By commenting out the first plugin and uncommenting the second, a completely different library is used without having to change a single line of code in the function.

The library is hidden behind an interface!

All good news?

As long as you are writing small scripts for prototyping, there is little need for this kind of abstraction. But coupled library code will lead to problems when existing code needs to be changed or libraries need to be swapped for others.

More experienced developers will immediately recognize how this kind of abstraction allows to maintain such code in future scenarios.

The end

You have now learned 3 common problems with 3 common solutions. But there is one thing you need to know. What I showed you here is just one way of doing things. Software development techniques always depend on the project and tools you use.

And even if you find the perfect solution today, it might be different tomorrow. That happens to me all the time and when I look back at old code, I often think: "What idiot wrote this code?"

So I will keep this eBook up-to-date with new findings and perhaps I will add more problems and solutions. So I invite you to come back here in the future and check for the latest version!

<https://pythonforeveryone.com/se>



www.pythonforeveryone.com