**The single responsibility principle**

The single responsibility principle (SRP) instructs developers to write code that has one *and only one* reason to change. If a class has more than one reason to change, it has more than one responsibility. Classes with more than a single responsibility should be broken down into smaller classes, each of which should have only one responsibility and reason to change.

This article explains that process and shows you how to create classes that have only a single responsibility but are still useful. Through a process of delegation and abstraction, a class that contains too many reasons to change should delegate one or more responsibilities to other classes.

It is difficult to overstate the importance of delegating to abstractions. It is the lynchpin of adaptive code and, without it, developers would struggle to adapt to changing requirements in the way that Scrum, Kanban, and other Agile frameworks demand.

**Problem statement**

To better explain the problem with having classes that hold too many responsibilities, this section explores an example. Listing 1 shows the TradeProcessor class. This class reads records from a file and updates a database.

Despite its small size, this sort of code is common and often needs to cope with new features and changes to requirements

**FIGURE 1** TradeProcessor

This is more than an example of a class that has too many responsibilities; it is an example of a single *method* that has too many responsibilities. By reading the code carefully, you can discern what this class is trying to achieve:

1. It reads ever line from a stream parameter, storing each line in a list of strings.
2. It parses out individual fields from each line and stores them in a more structured list of TradeRecord instances.
3. The parsing includes some validation and some logging to the console.
4. Each TradeRecord is saved by calling a procedure to insert trades into a database.

The responsibilities of the TradeProcessor are reading streams, parsing strings, validating fields, logging, and database insertion. The single responsibility principle states that this class, like all others, should have only a single reason to change. However, the reality of the TradeProcessor is that it will change under the following circumstances:

* When you decide not to use a Stream for input but instead read the trades from a remote call to a web service.
* When the format of the input data changes, perhaps with the addition of an extra field indicating the broker for the transaction.
* When the validation rules of the input data change.
* When the way in which you log warnings, errors, and information changes. If you are using a hosted web service, writing to the console would not be a viable option.
* When the database changes in some way—perhaps the insert\_trade stored procedure requires a new parameter for the broker, too, or you decide not to store the data in a relational database and opt for document storage, or the database is moving behind a web service that you must call.

For each of these changes, this class would have to be modified. Furthermore, unless you maintain a variety of versions, there is no possibility of adapting the TradeProcessor so that It is able to read from a different input source, for example. Image the maintenance headache when you are asked to add the ability to store the trades in a web service, but only if a certain command-line argument was supplied.

**Refactoring for clarity**

The first task on the road to refactoring the TradeProcessor so that it has one reason to change is to split the process\_trades method into smaller methods so that each one focuses on a single responsibility. Each of the following listings shows a single method from the refactored TradeProcessor class followed by an explanation of the changes.

First, Listing 2 shows the process\_trades method, which now does nothing more than delegate to other methods.

**LISTING 2** The process\_trades method is very minimal because it delegates work to other methods.

The original code was characterized by three distinct parts of a process—reading the trade data from a stream, converting the string data in the stream to TradeRecord instances, and writing the trades to persistent storage. Note that the output from one method feeds into the input to the next method. You cannot call \_\_store\_trades until you have the trade records returned from the \_\_parse\_trades method, and you cannot call \_\_parse\_trades until you have the lines returned from the \_\_read\_trade\_data method.

Taking each of these methods in order, let’s look at \_\_read\_trade\_data, in Listing 3.

**LISTING 3** \_\_read\_trade\_data

This code is preserved from the original implementation of the process\_trades method. It has simply been encapsulated in a method that returns the resultant string data as a tuple of strings. Note that this makes the return value read-only, whereas the original implementation unnecessarily allowed subsequent parts of the process to add further lines.

The \_\_parse\_trades method, shown in Listing 4, is next. It has changed somewhat from the original implementation because it, too, delegates some tasks to other methods.

**LISTING 4** \_\_parse\_trades delegates to other methods to limit its complexity.

This method delegates validation and mapping responsibilities to other methods. Without this delegation, this section of the process would still be too complex, and it would retain too many responsibilities. This \_\_validate\_trade\_data method, shown in Listing 5, returns a Boolean value to indicate whether any of the fields for a trade line are invalid.

**LISTING 5** All of the validation code is a single method.

The only change made to the original validation code is that it now delegates to yet another method for logging messages. Rather than embedding calls to the print function where needed, the \_\_log\_message method is used, shown in Listing 6.

**LISTING 6** The \_\_log\_message method is currently just a synonym for print.

Returning up the stack to the \_\_parse\_trades method, Listing 7 shows the other method to which it delegates. This method maps an array of strings representing the individual fields from the stream to an instance of the TradeRecord class.

**LISTING 7** Mapping from one type to another is a separate responsibility.

The sixth and final new method introduced by this refactor is \_\_store\_trades, shown in Listing 8. This method wraps the code for interacting with the database. It also delegates the information log message to the aforementioned \_\_log\_message method.

**LISTING 8** Now that the \_\_store\_trades method is in place, the responsibilities in this class are clearly demarcated.

Looking back at this refactoring, you can see that it is a clear improvement on the original implementation. However, what have you really achieved? Although the new process\_trades method is indisputably smaller than the monolithic original, and the code is definitely more readable, you have gained very little by way of adaptability. You can change the implementation of the \_\_log\_message method so that it, for example, writes to a file instead of to the console, but that involves a change to the TradeProcessor class, which is precisely what you want to avoid.

This refactoring has been an important steppingstone on the path to truly separating the responsibilities of this class. It has been a refactoring for clarity, not for adaptability. The next task is to split each responsibility into different classes and place them behind the interfaces. What you need is true abstraction to achieve useful adaptability.

Before you read on, if you are not familiar with interfaces in Python, read this article that explains this common computer programming concept (in other languages at least) for Python. <https://realpython.com/python-interface/>

**Refactoring for abstraction**

Building on the new TradeProcessor implementation, the next refactor introduces several abstractions that will allow you to handle almost any change request for this class. Although this running example might seem very small, perhaps even insignificant, it is a workable contrivance for the purposes of this tutorial. Also, is it *very* common for a small application such as this to grow into something much larger. When a few people start to use it, the feature requests begin to increase.

Often, the terms *prototype* and *proof of concept* are applied to such allegedly small applications, and the conversion from prototype to production application is relatively seamless. This is why the ability to refactor toward abstraction is such a touchstone of adaptive development. Without it, the myriad requests devolve into a “big ball of mud”—a class, or a group of classes in an assembly, with little delineation of responsibility and no discernible abstractions. The result is an application that has no unit tests and that is difficult to maintain and enhance, and yet that could be a critical piece of the line of business.

The first step in refactoring the TradeProcessor for abstraction is to design the interface of interfaces that it will use to perform the three high-level tasks of reading, processing, and storing the trade data. Figure 1 shows the first set of abstractions.

|  |  |  |
| --- | --- | --- |
|  |  | +get\_trade\_data() -> List[str]  < <INTERFACE> >  ITradeDataProvider |
| -tradeDataProvider  -tradeParser  -tradeStorage  +process\_trades() -> None  TradeProcessor |  | +parse(Iterable[str]) -> [TradeRecord]  < < INTERFACE> >  ITradeParser |
|  |  | < <INTERFACE> >  ITradeStorage  +persist(Iterator[TradeRecord] -> None |

**FIGURE 1** The TradeProcessor will now depend on three new interfaces.

Because you moved all the code from ProcessTrades into separate methods in the first refactor, you should have a good idea of where the first abstractions should be applied. As prescribed by the single responsibility principle, the three main responsibilities will be handled by different classes. You should not have direct dependencies from one class to another but should instead work via interfaces. Therefore, the three responsibilities are factored out into three separate interfaces. Listing 9 shows how the TradeProcessor class looks after this change.

**LISTING 9** The TradeProcessor is now the encapsulation of a process, and nothing more.

The class is now significantly different from its previous incarnation. It no longer contains the implementation details for the whole process but instead contains the *blueprint* for the process. The class models the process of transferring trade data from one format to another. This is its only responsibility, its only concern, and the only reason that this class should change. If the process itself changes, this class with change to reflect it. But if you decide you no longer want to retrieve data from a Stream, log to the console, or store the trades in a database, this class remains as is.

The interfaces that the TradeProcessor now depends on all live in a separate assembly. This ensures that neither the client nor the implementation assemblies reference each other. Also separated into another assembly are the three classes that implement these interfaces, the StreamDataProvider, SimpleTradeParser, and SqlAlchemyTradeStorage classes. Note that there is a naming convention used for these classes. First, the prefixed I was removed from the interface name and replaced with implementation-specific context that is required of the class. So StreamTradeDataProvider allows you to infer that it is an implementation of the ITradeDataProvider interface that retrieves its data from a Stream object. The SqlAlchemyTradeStorage class uses SqlAlchemy to persist the trade data. I have prefixed the ITradeParser implementation with the word Simple to indicate that is has no dependency context.

All three of these implementations are able to live in a single assembly due to their shared dependencies—core assemblies of the Microsoft .NET Framework. If you were to introduce an implementation that required a third-party dependency, a first-party dependency of your own, or a dependency from a non-core .NET Framework class, you should put these implementations into their own assemblies. For example, if you were to use the Dapper mapping library instead of SqlAlchemy, you would create an assembly called Services.Dapper, inside of which would be an ITradeStorage implementation called DapperTradeStorage.

The ITradeDataProvider interface does not depend on the Stream class. The previous version of the method for retrieving trade data required a Stream instance as a parameter, but this artificially tied the method to a dependency. When you are creating interfaces and refactoring towards abstraction, it is important that you do not retain dependencies where doing so would affect the adaptability of the code. The possibility of retrieving the trade data from sources other than a Stream has already been discussed, so the refactoring has ensured that this dependency is removed from the interface. Instead, the StreamTradeProvider requires a Stream as a constructor parameter, instead of a method parameter. By using the constructor, you can depend on almost anything without polluting the interface. Listing 10 shows the StreamTradeDataProvider implementation.

**LISTING 10** Context can be passed into classes via constructor parameters, keeping the interface clean.

Remember that the TradeProcessor class, which is the client of this code, is aware of nothing other than the get\_trade\_data method’s signature via the ITradeDataProvider. It has no knowledge whatsoever of how the real implementation retrieves the data—nor should it.

There are more abstractions that can be extracted from this example. Remember that the original parse\_trades method delegated responsibility for validation and for mapping. You can repeat the process of refactoring so that the SimpleTradeParser class does not have more than one responsibility. Figure 2 shows in Unified Markup Language (UML) how this can be achieved.

|  |  |  |
| --- | --- | --- |
| < <INTERFACE> >  ITradeParser  +parse(List[str]) -> List[TradeRecord] |  |  |
|  |  |  |
| SimpleTradeParser  -tradeValidator  -tradeMapper  +parse(List[str])) -> List[TradeRecord] |  | +map(List[str]) -> TradeRecord  < < INTERFACE> >  ITradeMapper |
|  |  | < <INTERFACE> >  ITradeValidator  +validate(List[str]) -> bool |

**FIGURE 2** The SimpleTradeParser is also refactored to ensure that each class has a single responsibility.

This process of abstracting responsibilities into interfaces (and their accompanying implementations) is recursive. As you inspect each class, you must determine the responsibilities that it has and factor them out until the class has only one. Listing 11 shows the SimpleTradeParser class, which delegates to interfaces where appropriate. Its single reason for change is if the overall structure of the trade data changes—for instance, if the data no longer uses comma-separated values and changes to using tabs, or perhaps XML.

**LISTING 11** The algorithm for parsing trade data is encapsulated in ITradeParser implementations.

The final refactor aims to abstract logging from two classes. Both the ITradeValidator and ITradeStorage implementations are still logging directly to the console. This time, instead of implementing your own logging class, you will create an adapter for the popular logging library, Log4Net. The UML class diagram in Figure 3 shows how this all fits together.

|  |  |  |
| --- | --- | --- |
| < <INTERFACE> >  ITradeStorage  +persist(List[TradeRecord]) -> None |  | +validate(List[str]) -> bool  < < INTERFACE> >  ITradeValidator |
|  |  |  |
| -logger  +persist(List[TradeRecord]) -> None  SimpleTradeStorage |  | SimpleTradeValidator  -logger  +validate(List[str]) -> bool |
|  |  |  |
| +LogWarning()  +LogInfo()  +LogError()  < <INTERFACE> >  ILogger |  |  |
|  |  |  |
| -log  +LogWarning()  +LogInfo()  +LogError()  Log4NetLoggerAdapter |  | +LogWarning()  +LogInfo()  +LogError()  < <INTERFACE> >  Log4Net.ILog |

**FIGURE 3** By Implementing an adapter for Log4Net, you need not reference it in every assembly.

The net benefit of creating an adapter class such as Log4NetLoggerAdapter is that you can convert a third-party reference into a first-party reference. Notice that both SqlAlchemyTradeStorage and SimpleTradeValidator both depend on the first-party ILogger interface. But, at run time, both will actually use Log4Net. The only references needed to Log4Net are in the entry point of the application and the newly created Service.Log4Net assembly. Any code that has a dependency on Log4Net, such as custom appenders, should line in the Service.Log4Net assembly. For now, only the adapter resides in this new assembly.

The refactoring validator class is shown in Listing 12. It now has no reference whatsoever to the console. Because of the flexibility of Log4Net, you can actually log to almost anywhere now. Total adaptability has been achieved as far as logging is concerned.

**LISTING 12** The SimpleTradeValidator class after refactoring

At this point, a quick recap is in order. Bear in mind that you have altered nothing as far as the functionality of the code is concerned. This should be proven by passing the Golden Master test, which is protecting against refactoring mistakes. Functionally, this code does exactly what it used to do. However, if you wanted to enhance it in any way, you could do so with ease. The added ability to adapt this code to a new purpose more than justifies the effort expended to refactor it.

Tip: When refactoring in this way, ensure that new code is written in a test-first fashion so that the Golden Master test is not the only test coverage that is in place.

Referring back to the original list of potential enhancements to this code, this new version allows you to implement each one without touching the existing classes.

* Request: You decide not to use a Stream for input but instead read the trades from a remote call to a web service.
  + Solution: Create a new ITradeDataProvider implementation that supplies the data from the service.
* Request: The format of the input data changes, perhaps with the addition of an extra field indicating the broker for the transaction.
  + Solution: Alter the implementation for the ITradeDataValidator, ITradeDataMapper, and ITradeStorage interfaces, which handle the new broker field.
* Request: The validation rules of the input data change.
  + Solution: Edit the ITradeDataValidator implementation to reflect the new rule changes.
* Request: The way in which you log warnings, errors, and information changes. If you are using a hosted web service, writing to the console would not be a viable option.
  + Solution: As discussed, Log4Net provides you with infinite options for logging, by virtue of the adapter.
* Request: The database changes in some way—perhaps the insert\_trade stored procedure requires a new parameter for the broker, too, or you decide not to store the data in a relational database and opt for document storage, or the database is moving behind a web service that you must call.
  + Solution: If the stored procedure changes, you would need to edit the SqlAlchemyTradeStorage class to include the broker field. For the other two options, you could create a MongoTradeStorage class that uses MongoDB to store the trades, and you could create a WebServiceTradeStorage class to hide the implementation behind a web service.

I hope you are now somewhat convinced that a combination of abstracting via interfaces, decoupling assemblies, aggressive refactoring, and adhering to the single responsibility principle are the foundation of adaptive code.

When you arrive at a scenario in which you code is neatly delegating to abstractions, the possibilities are endless. The rest of this chapter concentrates on other ways in which you can focus on a single responsibility per class.

**Conclusion**

The single responsibility principle has a hugely positive impact on the adaptability of code. Compared to equivalent code that does not adhere to the principle, SRP-compliant code leads to a greater number of classes that are smaller and more directed in scope. Where there would otherwise have been a single class or suite of classes with interdependencies and confusion of responsibility, the SRP introduces order and clarity.

The SRP is primarily achieved through abstracting code behind interfaces at run time. Some design patterns are excellent at supporting efforts to strictly regiment the SRP—in particular, the Adapter pattern and the Decorator pattern. The former enables much of your code to maintain first-party references to interfaces under your direct control, although in reality using a third-party library. The latter can be applies whenever some of a class’s functionality needs to be removes but it is too tightly coupled with the intent of the class. To stand alone.

What this chapter did not cover is how all these classes are orchestrated at run time. Passing interfaces into constructors was taken for granted in this article, but the next article will describes a variety of ways in which this can be accomplished.