

Single Responsibility Principle

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

NEW Forum Discussions: NEW

What Design Pattern Should I Choose?

Design Principles and Patterns Single Responsibility Principle

Motivation

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

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

A class should have only one reason to change. Example

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

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

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

// single responsibility principle - bad example

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

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

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

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

// single responsibility principle - good example

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

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

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

```
}
```

Conclusion

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