S.O.L.I.D. for your language

Stefan Roock (stefan.roock@it-agile.de)

Handouts for the SOLID session at Sokrates Germany 01.-03.-sep-2011.

It is pretty clear what SOLID means for class-typed object-oriented languages. For the other language types we collected some ideas but don't claim to have the final answer.

Example code for the SOLID principles in different programming languages is avaiable via GitHub: https://github.com/stefanroock/SOLID-for-dynamic-and-functional-languages--Sokrates-Germany-2011-

Table of Contents

Introduction

This paper starts with a brief description of the SOLID principles. Then the principles are applied to four different types of programming languages:

1. *class-typed object-oriented* languages like Java, C++ or C#
    a. programs have states
    b. states are encapsulated in objects that are instances of classes
    c. a class may inherit from another class
    d. polymorphism is bound to inheritance between classes
    e. for the context of this paper it doesn't matter if the type is checked during compile time or run time
2. *duck-typed object-oriented* languages like Ruby, Smalltalk or Python
    a. programs have states
    b. states are encapsulated in objects that are objects of classes
    c. objects are instances of classes
    d. a class may inherit from another class
    e. type checking doesn't check if an object is an instance of a certain class (has a specific type) like in class-typed object-oriented langauges; it is sufficient when a called method is available at the object at hand (so called "duck typing")
3. *typed functional* languages like Miranda, Scala, F# or Haskel
    a. programs are designed stateless (variables don't change their values over time)
    b. programs are made of functions
    c. types are explicitly defined (e.g. by Abstract Data Types)
4. *untyped functional* languages like Clojure, Scheme or Common Lisp
    a. programs have no state that might change over time
    b. programs are made of functions
    c. types are not explicitly defined
    d. for the context of this paper it doesn't matter if the type is checked during compile time or run time

A lot of the mentioned programming languages are hybrid. Scala has a class concept, in Python there can be free functions that are not bound to classes and most functional languages have concepts for handling state. For the context of this paper we only use the "pure" concepts of the programming languages.

The goal of the workshop is to gain a better understanding of the SOLID principles and there general applicability under certain programming paradigms.

The S.O.L.I.D. principles

SOLID is an acronym for five software design principles defined by Robert C. Martin (aka Uncle Bob):

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle (SRP)

“A module should have one, and only one, reason to change.”

Rationale

The SRP aims at reducing the costs of changing existing requirements. There should be one place in the source code that has to be modified. And at this place there should only be the code for this requirement and not the code of other requirements so that unwanted side-effects are avoided. By separating different responsibilities into different code units it is easier to reuse existing code, too.

So the effects of the SRP are:

- avoid unwanted side-effects when modifying existing requirements
- minimize the number of source codes that have to be touched for a modification
- increase reusability

Open Closed Principle (OCP)

“A module should be open for extension but closed for modification.”

Rationale

This principle is about the handling of change for ***new*** requirements. If there are new requirements, there should be no need to modify running, existing code. Instead, new requirements should be implemented by new code. OCP correlates with the DIP, since the openness for change is often reached by the inversion of dependencies. It is a balanced design decision, in which way a system is open or closed. It is not possible to design a system, which is open for all kinds of new requirements.

The effects of the OCP are:

- avoid unwanted side effects when implementing new requirements
- minimize the amount of source codes that have to be touched for a new requirement

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

The formal definition of the LSP is: *“Let q(x) be a property provable about objects x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T.”* [5]

For class-typed object-oriented languages this means that inheritance is not really an **is-a** relation. A square **is a** rectangle but a square class inheriting from a rectangle class breaks the LSP. Think about a method set_side_lengths(int, int) defined in the rectangle class. This methods would have been redefined in the square subclass in a way that would break the LSP. With the LSP in mind inheritance is more a **behaves-like-a** relationship.

Rationale

The LSP ensures that type hierarchies are well-formed. In class-typed object-oriented languages the LSP ensures that the class hierarchy defines a valid type hierarchy. When the LSP is violated often the OCP is violated as a collateral damage.

The effect of the LSP is:

- avoid unwanted side effects when implementing new requirements

Interface Segregation Principle (ISP)

“Many client specific interfaces are better than one general purpose interface.”

Rationale

The ISP reduces the coupling within the system. Reduced coupling reduces unwanted side effects and increases reusability.

The effects of the ISP are:

- avoid unwanted side effects
- increase reusability

Dependency Inversion Principle (DIP)

“Depend upon abstractions. Do not depend upon concretions.”

Rationale

This principle is about the handling of change in ***existing*** requirements. If details change, there will be no need to change the rest of the system. DIP correlates with OCP, since there is the chance, that new requirements could be satisfied by adding new details for existing abstractions.

The effects of the DIP are:

- reduce efforts for adapting existing code when modifying existing code
- avoid unwanted side effects
- increase reusability

References

- [1] Wikipedia: http://en.wikipedia.org/wiki/SOLID_(object-oriented_design)
- [2] Robert C. Martin: “Design Principles and Design Patterns”: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- [3] Robert C. Martin: “Agile Software Development. Principles, Patterns, and Practices”
- [4] Wikipedia: http://en.wikipedia.org/wiki/Liskov_substitution_principle
- [5] Barbara Liskov, Jeannette Wing, “A behavioral notion of subtyping”, ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6 (November 1994), pp. 1811 – 1841.
- [6| Bertrand Meyer: “Tell Less, Say More - The Power of Implicitness”, http://se.ethz.ch/~meyer/publications/computer/implicitness.pdf

S.O.L.I.D. for class-typed OO languages (like Java, C#, C++)

Single Responsibility Principle (SRP)

“A module should have one, and only one, reason to change.”

Applied to class-typed object-oriented languages a *module* is defined by

- classes
- methods
- packages

Open Closed Principle (OCP)

“A module should be open for extension but closed for modification.”

In class-typed object-oriented languages a module in the context of OCP means a class in most cases. There are several possibilities to implement the OCP:

- inherit from an existing class
- provide a plugin in an existing class, e.g. with the Strategy design pattern
- create a new class and delegate to an existing class

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

For class-typed object-oriented languages the LSP just defines the constraints that have to hold true when inheriting from a class or implementing an interface. When - for example - a client A expects that a method M of a class B never throws an exception, subclasses of B should not throw exceptions.

Thesis

The LSP is defined in a formal way but the core really is what programmers expect. Let’s take the collection library of the JDK. In the interface *Collection* there is a method *remove(Object)*. In the documentation it is stated that *remove* my throw an *UnsupportedOperationException*. This happens for example when the collection at hand is unmodifyable. Formally the LSP holds true here but in most cases programmers don’t expect this exception and write code like this:

```
public void foo(Collection c) {
    Object toRemove = findObjectToRemove();
    c.remove(toRemove);
}
```

According to the *Collection* interface the “correct” implementation would be something like:

```
public void foo(Collection c) {
    Object toRemove = findObjectToRemove();
    try {
        c.remove(toRemove);
    } catch (UnsupportedOperationException e) {
        // do something clever
    }
}
```

The problem here is that it is very hard to come up with something clever to do when catching the exception. Therefore programmers decide to ignore the exception and hope that it won't occur. Creating an implementation of the *Collection* interface that throws an exception in remove violates the expectation of the client code and therefore breaks the LSP.

Interface Segregation Principle (ISP)

"Many client specific interfaces are better than one general purpose interface."

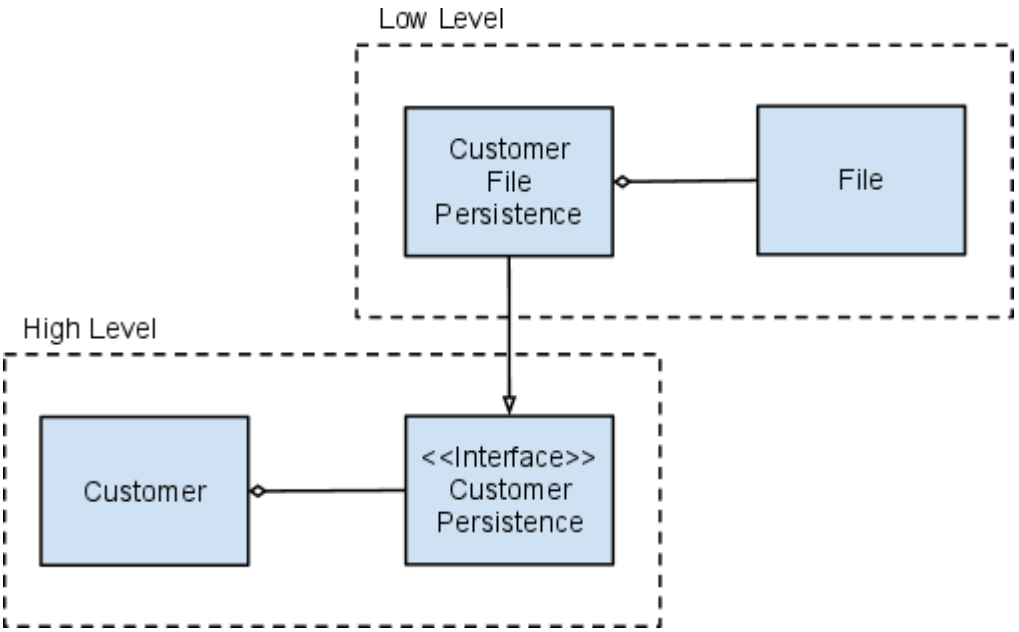
There are at least two possible implementations of the ISP:

- create an interface for every client and let the supplier class implement all the interfaces
- create a class for every client with its own interface

Dependency Inversion Principle (DIP)

"Depend upon abstractions. Do not depend upon concretions."

The class-typed object-oriented languages the DIP is mainly applied to classes. So a high level conceptual class like *Customer* should not be dependent on a low level implementation class like *File*. Normally the principle is implemented by creating an interface with the methods required by the client (e.g. *CustomerPersistence*) and using an adapter to connect the low level implementation to it (e.g. *CustomerFilePersistence*).



By enforcing the definition of interfaces from the client perspective the DIP correlates with the ISP. By decoupling high level concepts and low level implementations the DIP supports the OCP principle.

Example (from [1])

Car has Carburator and Transmission.
Car.accelerate calls Carburator.increase_fuel_to_air_ratio and probably some more things.

This design violates the DIP. You would have to change the Car class if you gave a Diesel or even an electric engine were increasing the fuel to air ratio makes no sense.

According to the DIP the problem here is that the more abstract concept Car depends on the more concrete implementations of Carburator and Transmission. It shouldn't.

A better design would be to introduce an abstract Engine concept with a method increase_power. The Carburator would implement the Engine concept and the Car would use the Engine and not the concrete Carburator implementation. If we would need a Diesel we simply would add a Diesel engine implementation of the Engine concept and won't have to change the Car.

References

- [1] The car example in the DIP section was used by Robert Martin in 1992 according to http://www.mbohlen.de/2009/11/09/welcome-back-uncle-bob-oop-2010/

S.O.L.I.D. for duck-typed OO languages

(like Ruby, Smalltallk, Python)

Single Responsibility Principle (SRP)

“A module should have one, and only one, reason to change.”

Thesis

Since the principle doesn’t touch types at all it is applicable and relevant to duck-typed object-oriented languages as it is to class-typed object-oriented languages.

Open Closed Principle (OCP)

“A module should be open for extension but closed for modification.”

Thesis (from [1])

The OCP is relevant and applicable to duck-typed object-oriented languages as it is to class-typed object-oriented languages. Since it is often possible to extend a class or object at runtime with additional methods there is at least one additional way to implement the OCP. Using the extension mechanism is in some aspects even more elegant than inheritance since you don’t have to care about the question who creates instances of which class.

Example of extending a class in Ruby at runtime:

```
# foo.rb
class Foo
    def method1 *args
        ...
    end
end

# foo2.rb
class Foo
    def method2 *args
        ...
    end
end

# my_file.rb
foo = Foo.new
foo.method1 :arg1, :arg2
foo.method2 :arg1, :arg2
```

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

Unlike in class-typed object-oriented languages in duck-typed object-oriented languages the inheritance relation between classes doesn’t imply a subtype relation. In duck-typed languages the class of an object isn’t relevant to typing at all. All that matters is that the called method is available at the object at hand.
There are no explicitly defined type interfaces. But there are implicitly defined interfaces. If a method m1 calls m2 and m3 on an object o1 it defines an implicit interface with its expectations about the behaviour of object o1.

Thesis 1 (from [2], [3])

The LSP applies to duck-typed languages as well. You have to ensure that every object you pass to a method **behaves as** the methods expects.

Thesis 2

Thesis 1 is right, but: In class-typed object-oriented languages there are two perspectives from which an interface may be defined. You can define it as a abstraction about existing classes: In the JDK the *Collection* interface is an abstraction about the collection classes. Or cou can define an interface from the perspective of the client expressing his expectations: In the JDK the *Comparable* interface is defined from the perspective of the sorted collection classes. It expresses what they need to sort the elements of the collection. With interfaces defined as an abstraction the LSP is far more relevant than with interfaces defined from the perspective of a client. Every new client would define a new interface and when creating new classes conforming to the interface you would test the new class against the client. Problems would become visible and fixed very early without even knowing about the LSP.
In duck-typed languages every (implicit) interface is defined from the perspective of the client. Therefore the LSP is applicable to duck-typed languages but rarely of of relevance to the daily work of a programmer.

Interface Segregation Principle (ISP)

“Many client specific interfaces are better than one general purpose interface.”

Thesis

Since in duck-typed object-oriented languages interfaces are defined implicitly be the client the ISP is enforced by the language and has little relevance for the daily work of programmers.

Antithesis

There is still value in creating smaller classes from the perspective of the client and therefore the ISP is relevant in duck-typed object-oriented languages.

Dependency Inversion Principle (DIP)

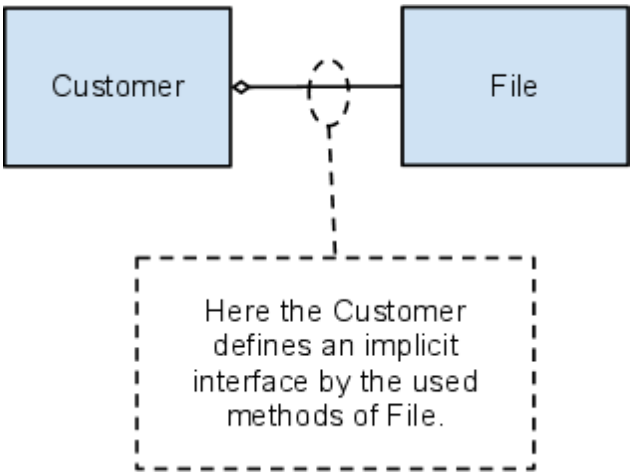
“Depend upon abstractions. Do not depend upon concretions.”

Thesis

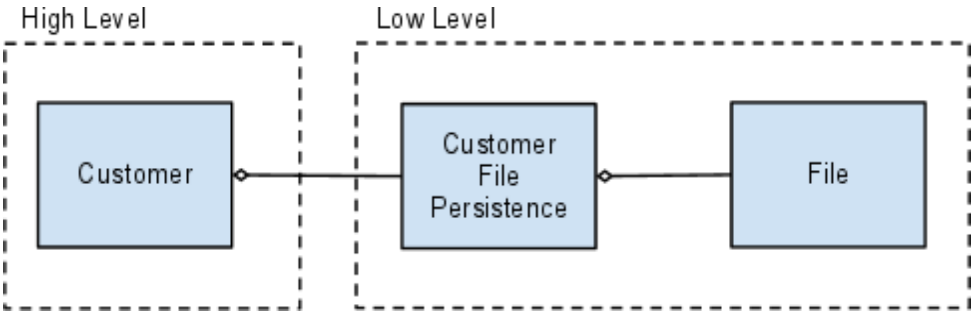
The DIP for duck-typed object-oriented languages is very similar to the DIP for class-typed object-oriented languages. Due to duck-typing the implementation of the DIP is very lightweight. The only adherence to the concrete type/class is during object creation. Therefore to follow the DIP in duck-typed object-oriented languages we just have to place the object creation outside the high level classes.

Antithesis

That would be possible on a syntactic level but it screws up on the semantic level. Let's take the *Customer* and *File* example. If you build it like proposed by the previous thesis you would end up with something like this:



Two types of problem may occur. If *File* is an existing class from a library it would force the *Customer* to use its wording for defining the implicit interface. So the implicit interface could end up with methods like *open_file*, *close_file*, *read_bytes*, *is_eof* which is low level on a semantic level. Or *File* is a class defined by ourself. In this case the interface defined implicitly be the Customer would be semantically valid and having methods like *save_customer* and *load_customer*. But with our simple design it would force the class *File* to have methods that we wouldn't expect in a *File* class.
Therefore we need some kind of adapter implementation in duck-typed object-oriented languages, too:



Anti-Antithesis

When you use dynamic class extension (see section about OCP in duck-typed object-oriented languages) you don't need the adapter class and could live with the simple first approach.

References

- [1] "The Open-Closed Principle for Languages with Open Classes": http://blog.objectmentor.com/articles/2008/09/04/the-open-closed-principle-for-languages-with-open-classes
- [2] The Liskov Substitution Principle for "Duck-Typed" Languages: http://blog.objectmentor.com/articles/2008/09/06/the-liskov-substitution-principle-for-duck-typed-languages
- [3] The Liskov Substitution Principle (LSP) in duck typed programming languages: http://stefanroock.wordpress.com/2010/11/08/the-liskov-substitution-principle-lsp-in-duck-typed-programming-languages/

S.O.L.I.D. for typed functional languages (like Scala, F#, Miranda, Haskel)

Single Responsibility Principle (SRP)

“A module should have one, and only one, reason to change.”

Thesis

In functional languages there are typically three types of modules:

1. functions
2. defined data structures (records, structs)
3. packages/modules of functions and/or data structures

The SRP applies to all of these.

Open Closed Principle (OCP)

“A module should be open for extension but closed for modification.”

Thesis (from [1])

The Open-Closed-Principle has inheritance as a core concept and therefore only has weak relevance to functional programming.

Antitheses 1 (from [1])

“Take for instance the Open/closed principle. I think anyone would rather deal with a library that is extensible by adding to it, rather than having to fiddle with it and modify it. In functional languages, one can argue that this is easy to achieve because of immutability, which makes it possible to use individual functions as basic building blocks, as opposed to classes, which often tend to be more complex than they should be.”

Antithesis 2

There are a lot of different possibilities to implement a solution conforming to the OCP. It is possibly via delegation without inheritance. Therefore there should be an application of the OCP to functional languages.

Question regarding Antithesis 2

How could such an application look like?

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

Thesis (from [1])

The LSP is all about inheritance that isn’t of that relevance in functional languages. The LSP is enforced by the language already. So the principle holds true but is of less relevance in the day to day work of functional programmers.

Antithesis

The LSP is not about inheritance but about subtyping. Subtyping is used explicitly or implicitly in most modern programming languages. But in some languages programmers don’t have to care that much about the LSP in their daily work. That they don’t have to care that much doesn’t depend on wether the language is functional or object-oriented but wether the language is class-typed or duck-typed (see argumentation at the section of duck-typed object-oriented languages).

Question regarding the Antithesis

What does that mean for typed functional languages?

Interface Segregation Principle (ISP)

“Many client specific interfaces are better than one general purpose interface.”

Thesis

Packages or modules with functions have a public interface. Therefore the ISP applies to functional languages as well.
Since functions are first class citizens in functional languages it may even apply to single functions.

Question regarding the Thesis

What does that mean regarding the implementation of the ISP to functions?

Dependency Inversion Principle (DIP)

“Depend upon abstractions. Do not depend upon concretions.”

Thesis (from [1])

Dependency inversion is not a relevant issue for typed functional languages as the language itself supports higher-order-functions to encapsulate this concept.

Antithesis

The DIP applies to typed functional languages but other idioms / design patterns are used for its implementation.

Question regarding the Antithesis

How can we apply the DIP to typed functional languages? Which idioms/design patterns would we use?

References

- [1] http://moiraesoftware.com/?p=434

S.O.L.I.D. for untyped functional languages (like Clojure, Scheme, Common Lisp)

Single Responsibility Principle (SRP)

“A module should have one, and only one, reason to change.”

Thesis

In functional languages there are typically three types of modules:

4. functions
5. defined data structures (records, structs)
6. packages/modules of functions and/or data structures

The SRP applies to all of these.

Open Closed Principle (OCP)

“A module should be open for extension but closed for modification.”

Thesis

There are a lot of different possibilities to implement a solution conforming to the OCP. It is possibly via delegation without inheritance. Therefore there should be an application of the OCP to functional languages.

Question regarding the Thesis

How could such an application look like? Would multimethods (Clojure) play a role?

Liskov Substitution Principle (LSP)

“Subclasses should be substitutable for their base classes.”

Thesis

The LSP is not about inheritance but about subtyping. Subtyping is used explicitly or implicitly in most modern programming languages. But in some languages programmers don’t have to care that much about the LSP in their daily work. That they don’t have to care that much doesn’t depend on wether the language is functional or object-oriented but wether the language is typed or duck types (see argumentation at the section of duck-typed object-oriented languages). Therefore the LSP is applicable to untyped functional languages but rarely relevant to the daily

work of a programmer.

Question regarding the Thesis

Do multimethods play a role here?

Interface Segregation Principle (ISP)

“Many client specific interfaces are better than one general purpose interface.”

Thesis

Packages or modules with functions have a public interface. Therefore the ISP applies to functional languages as well.
Since functions are first class citizens in functional languages it may even apply to single functions.

Question regarding the Thesis

What does that mean regarding the implementation of the ISP to functions? Would it be relevant to the daily work of a programmer?

Dependency Inversion Principle (DIP)

“Depend upon abstractions. Do not depend upon concretions.”

Thesis 1

The DIP applies to untyped functional languages but other idioms / design patterns are used for its implementation.
The DIP for functions is easy to adapt. High level functions should not call low level functions. Instead they should call the functions through abstractions. For untyped functional languages this is easy to implement: Just call a function that is provided as a parameter.

Question regarding Thesis 1

How would that look like in a concrete implementation?

Thesis 2

The DIP is applicable to data structures (lists, records) as well.

Acknowledgements

I appreciate the discussions with Steven Collins about this topic and the feedback I got from Ilja Preuss.

Design principles like SOLID are complemented by agile engineering practices like TDD, Pair Programming etc. Read about the agile engineering practices in (german):

Learn more about agile design at the

Certified Scrum Developer Training

http://www.it-agile.de/csd.html