SOLID

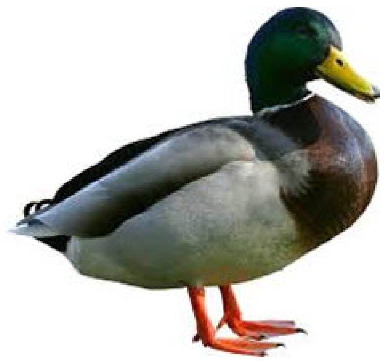
5 principles to guide and evaluate class design

Class design

The principles grew out of experience with systems implemented in languages like C++ and Java.

In languages like Python that feature *duck-typing*, some of the principles may seem less relevant because the "problem" addressed by the principle is not a problem in that language.

But keeping the intention of the principles in mind will still help you become a better designer.



What does this example mean?

"If it walks like a duck and quacks like a duck, it's a duck."

In fact, one of the principles will help refine our ideas about what it means to "*walk like a duck and quack like a duck*"

The SOLID principles

The Single Responsibility Principle	SRP
The Open Closed Principle	OCP
The Liskov Substitution Principle	LSP
The Interface Segregation Principle	ISP
The Dependency Inversion Principle	DIP

The Single Responsibility Principle SRP
- A class should have only one reason to change.

The Open Closed Principle OCP
- Software entities should be open for extension but closed for modification.

The Liskov Substitution Principle LSP
- Subtypes must be substitutable for their base types.

The Interface Segregation Principle ISP
- Clients should not be forced to depend on methods that they do not use.

The Dependency Inversion Principle DIP
- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

SOLID: SRP

Single Responsibility Principle: SRP

The Single Responsibility Principle

A class should have only one reason to change.

- Closely related to the concept of *cohesion*. In this case we don't want a change to affect unrelated code that happens to be in the same module.

  And if a module is changing, we don't want to be forced to *retest* code that is not related to that change.

- The SRP is independent of the type system and so applies for nominal typing, duck-typing, etc.

- The principle seems straightforward, but in practice we need a clearer understanding of *"reason to change".*

SRP: What is a responsibility?

- The best way to view a responsibility is a collection of functions in a class that serve *one particular user role*.
- Then we can view users in that role as the single source of change for that responsibility.
- For example:

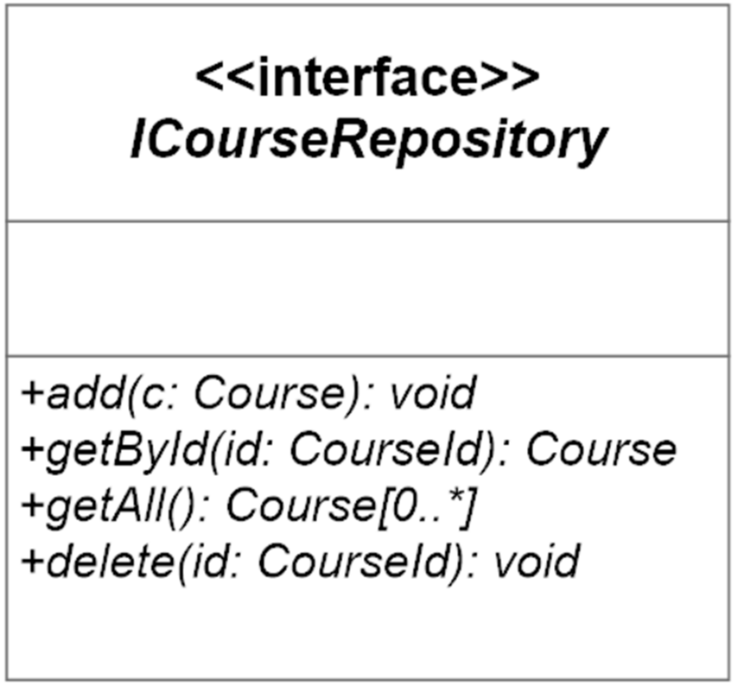
  Who is served by Persistence modules? The DBA.

  Who is served by a Shopping Cart? The customer.

  Who is served by Reports? Whoever uses the report.

SRP: Example 1

Does that mean ideal classes should do only one thing?


<<interface>>
ICourseRepository
+add(c: Course): void
+getById(id: CourseId): Course
+getAll(): Course[0..*]
+delete(id: CourseId): void

Classes that implement only this interface do four things, but they are all part of a single responsibility.

SRP: Example 2

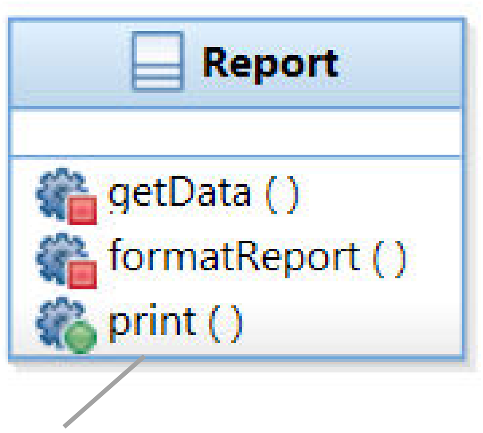
- Outline of extract from an automated code tester that alerts the relevant developer when a test fails.

```
class TestingService:

    def __init__(self, sms_messager, ...test details omitted... ):
        self.sms_messager = sms_messager

    # much code omitted

    def defect_alert(self, staff):
        self.sms_messager.send(mobile_num=staff.mobile_num, message=..
```

```
class TestingService:

    def __init__(self, messager, ...):
        self.messager = messager

    # much code omitted

    def defect_alert(self, staff):
        self.messager.send(staff, message)
```

Here the unrelated responsibility has been factored out. No longer vulnerable to changes in alert preferences, etc.

SRP: Example 3



many other responsibilities to do with reports here

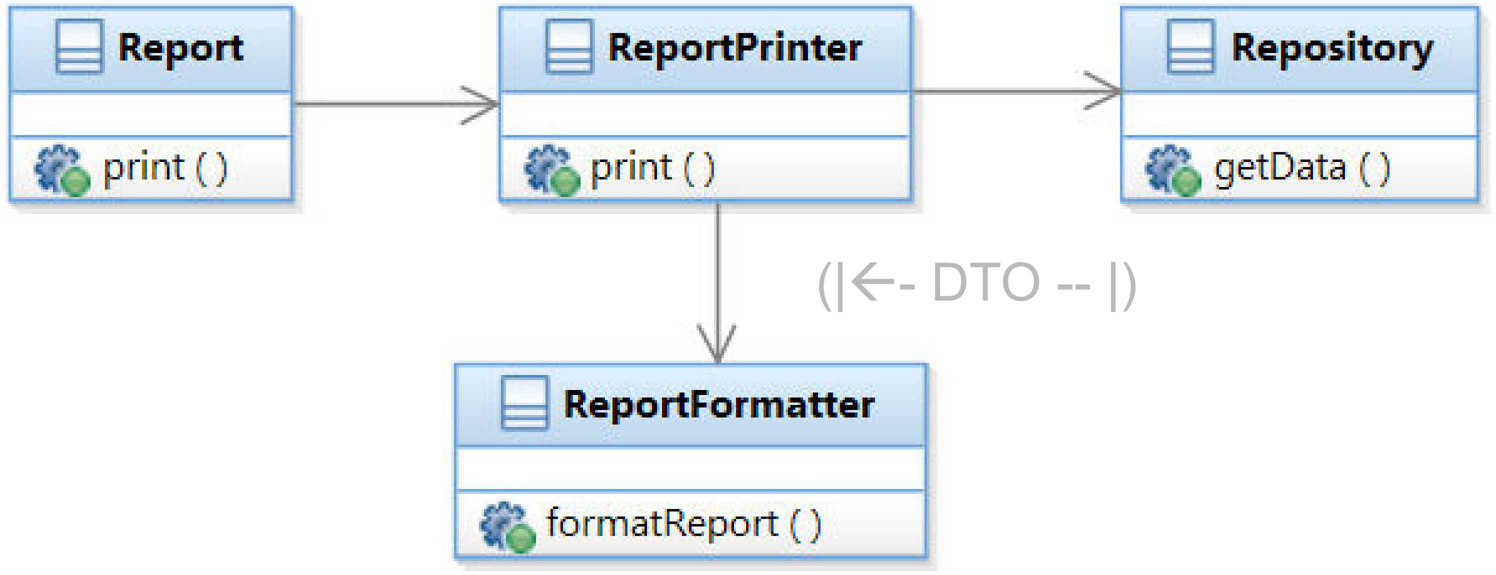
- Reporting is often mixed up with other logic. Here it has been separated out into a Report class, but it is responsible for handling the printing itself as well as other report responsibilities (not shown).
- To do that it needs to retrieve persisted data, format the data for presentation and then print it.
- Whenever:
  - users of the actual report change the way they want the data presented, or
  - when the DBAs change the persistence method, or
  - when the method of printing changes,

...then the class needs to change.

We need to split out the responsibilities.

SRP: Example 3 refactored

("refactoring" means improving a design without affecting external behaviour")



- Now each class has a single well-defined responsibility. The application is also easier to understand - it is clear where responsibilities live.

- These classes will have other methods and attributes, not shown here. But still they are quite small. Do we need to go so far?

SOLID Principles are not recipes

Need we go so far with the SRP? Many examples of the SRP on the web just keep making classes smaller and smaller. Do we stop only when a class has a single method?

When to apply the principle, when to stop?

- Informally, if you can't summarize what a class does without using "*and*" then it has more than one responsibility. But this is better done in terms of sources of change.

- If you start to reach the point of having tiny classes that do not abstract a complete concept or service, then those classes are not cohesive and you have probably over-applied the principle .

Don't forget cohesion

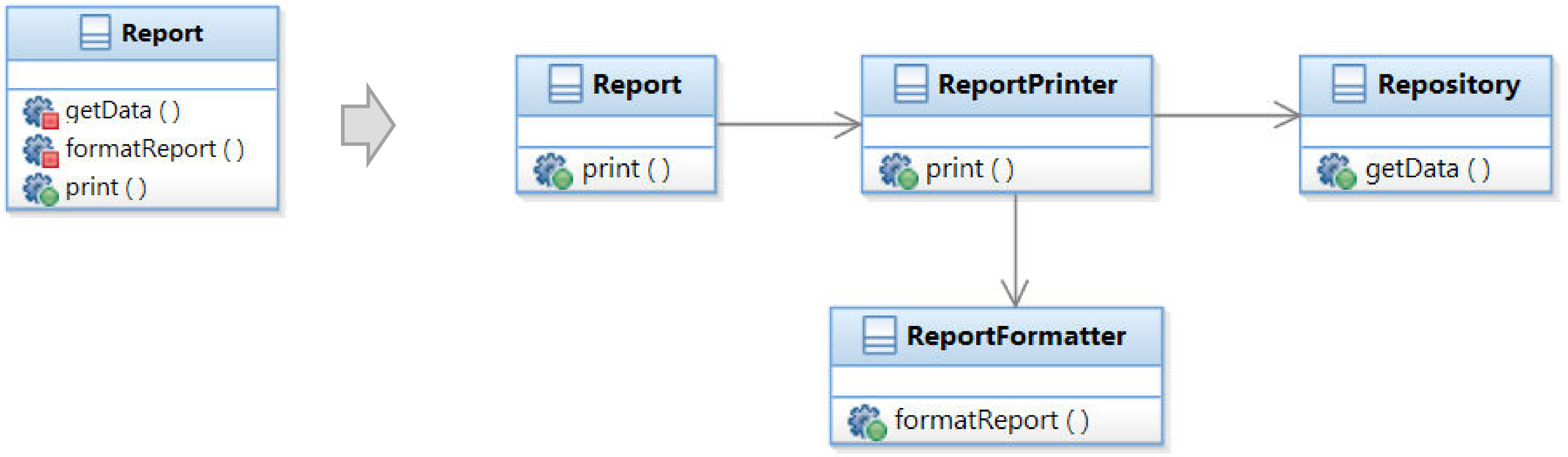
The problem is if you apply the SRP simplistically, your classes will get smaller and smaller and the code can become difficult to understand.

Don't forget the principles of good abstraction and cohesion that pull in the other direction. You need to find a balance between the two forces.

From the originator of the SRP:

> Gather together the things that change for the same reasons. Separate those things that change for different reasons.

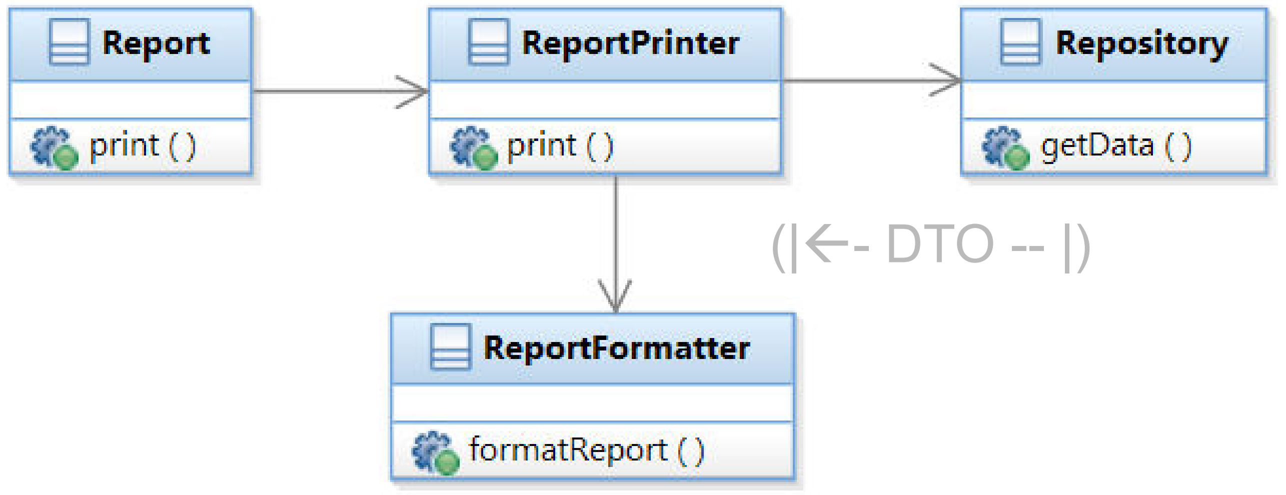
Don't forget the first part.



Q: Why does Report still need to look like it does printing and have a print() method?

A: In this case it is so that clients are not affected.
We are refactoring - no change to external behaviour

There is still a problem with this design



- What if we need to change the form of data source? Or the type of formatter?
- At the moment the *dependencies are fixed* and we will need to modify the code to make a change.
- How to fix? The next principle, OCP, will help.

SOLID: OCP

Open Closed Principle (OCP)

- Rigidity is a symptom of bad software design. Rigidity is the opposite of modifiability - the design is hard to change safely
- We know change is inevitable – so we want our software to be *stable* when we need to change it to support new requirements

We don’t want a single change to trigger chains of modifications as a result of chains of dependencies in our code.

We'd like to *extend* the behaviour of our systems *without changing* what is already tested and working successfully.

Open Closed Principle

Generally agreed that this principle is the key to the object-oriented approach. In its classic form it is just the application of polymorphism.

We want to implement new requirements by *adding/injecting* new code rather than by *changing* code that already works

Modules should be *open* for extension,
but *closed* for modification

Open for extension:

The behaviour of the module can be extended with new behaviours. We can change what it does.

Closed for modification:

We make those extensions without changing the code of the module. It remains untouched.

Open Closed Principle

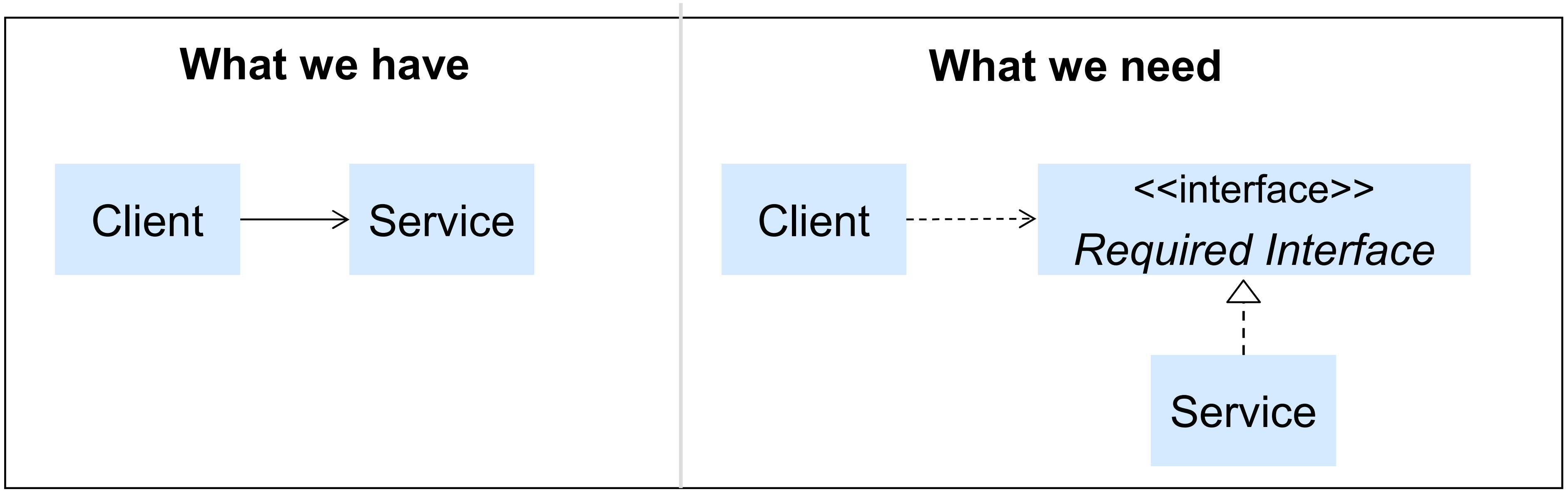
In the classic form we can add new behaviour by using inheritance and adding new derivative classes. Nothing else needs to be changed.

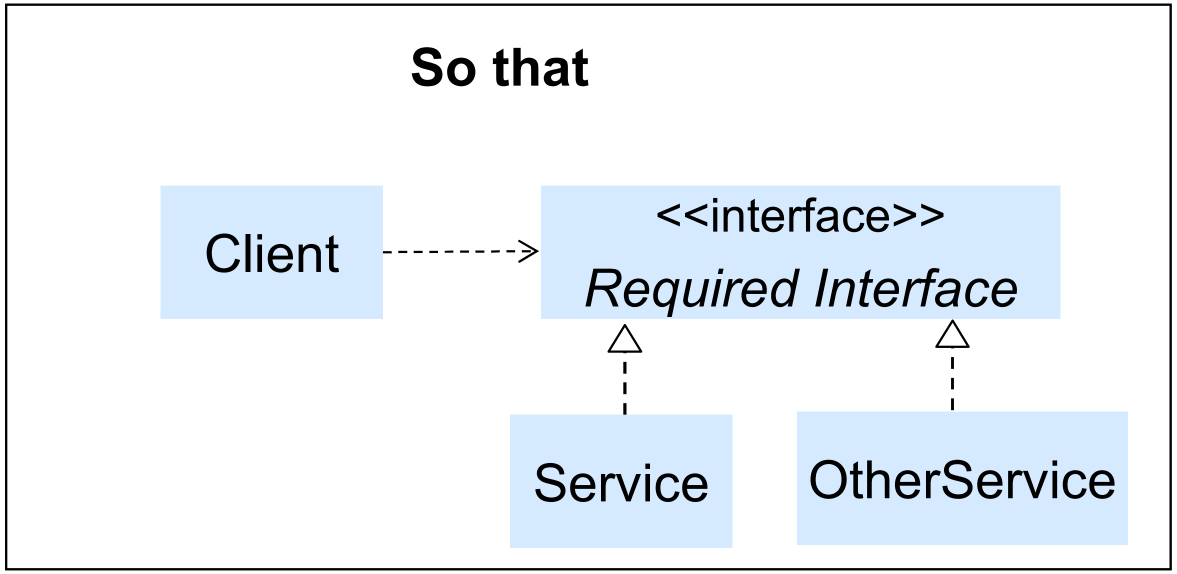
The current way it is applied is to design such that classes depend on abstractions rather than on other concrete classes.

Languages may provide other ways of applying the OCP. You've probably used lambdas or passed functions as arguments in Python.

It's still the same principle - inject new code rather than modifying existing code.

Open Closed Principle: Modern redefinition





What is an interface?

Abstractions everywhere?

- Does that mean we should create abstractions wherever possible?

  No, in many cases flexibility will not be needed at that particular point and so we are spending time and adding complexity for no benefit. YAGNI – no unnecessary complexity

  What is YAGNI here?

- One approach is to wait for change to highlight a point of volatility. Then add flexibility at that point to handle future changes of a similar kind.

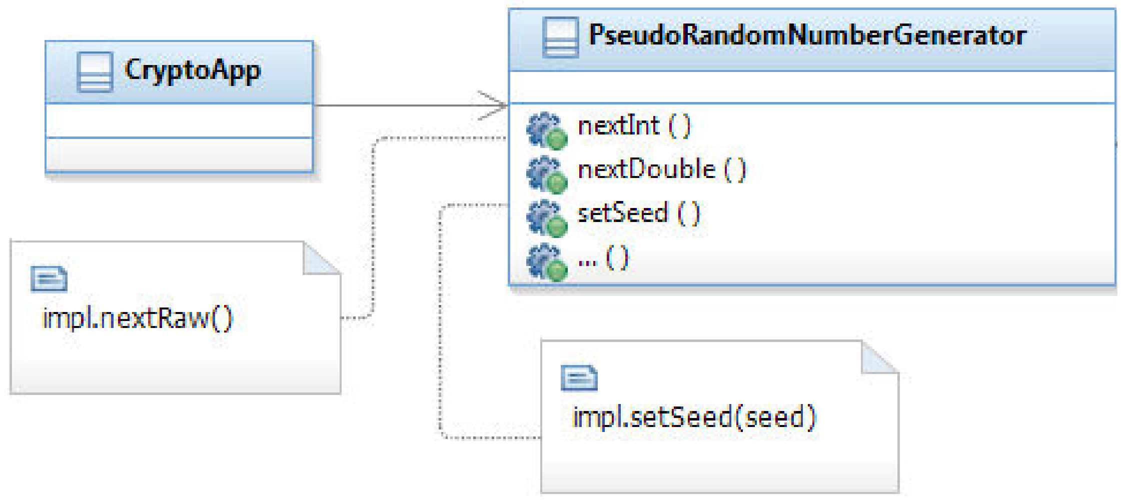
- Often the need for testability will point to where abstractions need to be built in.

- Iterative development helps us spot points of volatility early.

The principle is equally relevant in dynamic languages, although the need to open up existing code is much less.

Example

- Here we want flexibility to switch an algorithm at run-time.

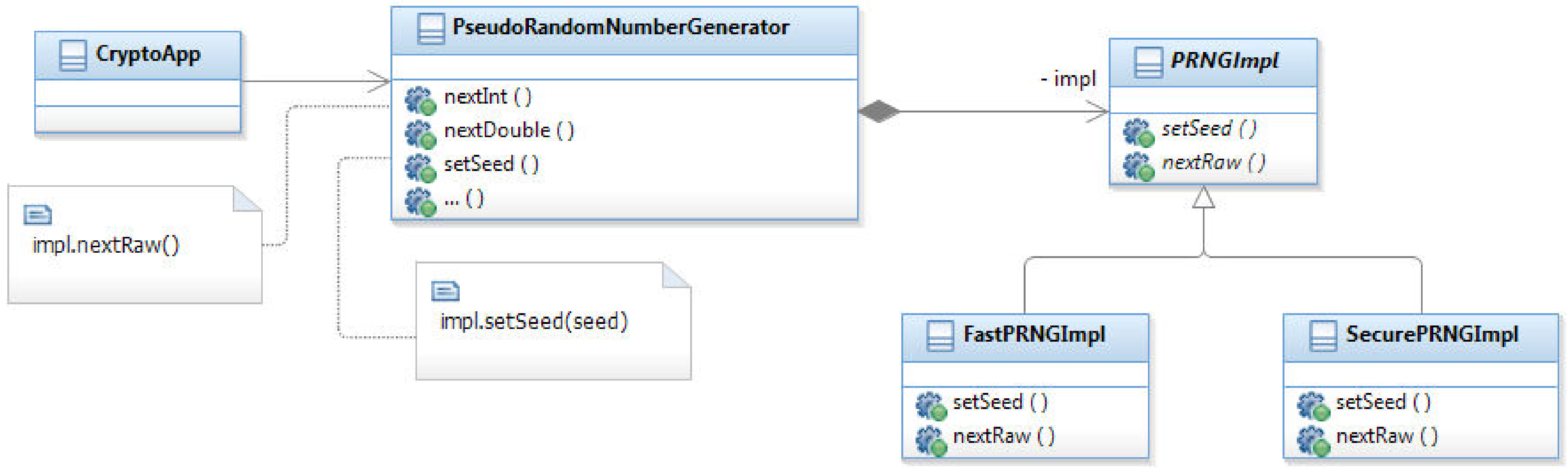


- It's a technique used often in design patterns to *"encapsulate the concept that varies".* The pattern on this slide is called the **Strategy Pattern**

- Testability: to test the PRNG in isolation we need to provide a mock PRNGImpl we can control from the tests. This is now easy - we just write a mock version that implements the interface and inject it.

Example

- Here we want flexibility to switch an algorithm at run-time.



- It's a technique used often in design patterns to *"encapsulate the concept that varies"*. The pattern on this slide is called the **Strategy Pattern**
- Testability: to test the PRNG in isolation we need to provide a mock PRNGImpl we can control from the tests. This is now easy - we just write a mock version that implements the interface and inject it.

SOLID: LSP

Liskov Substitution Principle: LSP

- Just saw that we can conform to the OCP through the mechanisms of abstraction and polymorphism. Inheritance allows a module, expressed in terms of a base type, to be extendable without modification

- But, for this to work reliably:

A short version of the Substitution Principle

Code that uses a reference to a base type must be able to use objects of derived types without knowing it:

Subtypes must be substitutable for their base types

Why? Because a client should be able to make reasonable assumptions about the behaviour of a type, and to depend on those assumptions.

LSP: quote from the original paper

What is wanted here is something like the following substitution property:

If for each object *o1* of type *S* there is an object *o2* of type *T* such that for all programs *P* defined in terms of *T*, *the behaviour of P is unchanged* when *o1* is substituted for *o2* then *S* is a subtype of *T*

Supertype

The contract

From earlier: A client should be able to make reasonable assumptions about the behaviour of a type, and to depend on those assumptions.

But how do you know what your clients might reasonably assume?

- This seems like a dangerous way to build systems
- We need some way of stating "*reasonable assumptions*" explicitly
- This is provided by the ***contract****:* the set of method preconditions and postconditions (including preservation of class invariants).

Modifying the contract of a method

A subtype method can *weaken* the precondition

A subtype method can *strengthen* the postcondition

Weaken means to *require less* from the caller than the supertype method requires

Strengthen means to *promise more* to the caller than the supertype method promises.

Example - one of Liskov's own examples

As an example, consider a family of integer set types

IntSet at the top of the hierarchy provides a minimal set of methods and subtypes may override these methods and may also provide additional methods.

IntSet has the following method to add 0 to the set.

```
def add_zero(self):
    #Pre:  the set is not empty
    #Post: set contains 0
```

Then if a subtype redefines the contract to be:

```
def add_zero(self):
    #Pre:
    #Post: set contains 0
```

This is *legal* since the precondition has been weakened

Example (cont)

IntSet has the following method to add a given int.

```
def add (self, i):
    #Post: set contains i
```

A *subtype LogIntSet* is an *IntSet* plus a log which is also a set. The Log contains all the ints that have ever been contained in the *IntSet*. The contract for the *add()* method of *LogIntSet* is:

```
def add(self, i):
    #Post: set contains i and log contains i
```

This is *legal* since the postcondition has been strengthened. It implies the supertype postcondition and promises something more as well.

Example (cont)

Now if we have a subtype, *OddIntSet*, with the following add method:

```
def add (self, i):
    #Post: set contains i if i is odd,
    #      otherwise the set is unchanged
```

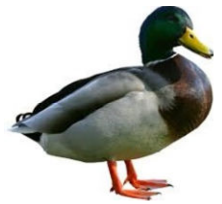
This is an *illegal* subtype method since the postcondition does not imply the supertype condition.

At some point a programmer might use an *OddIntSet* as an *IntSet* and the expectation will be that even values of i should be added to the set.

LSP in practice: *"Don't surprise people"*

- The issue comes down to programmers' expectations: "what they might reasonably assume"
- The key is to keep a contract in mind when subclassing.
- In languages with duck-typing the actual type is not important, what matters is that the method is available for the object it is called on.

The interface is implicit rather than explicit, but there are still expectations about behaviour. The contract is still there. Don't surprise.



If your object is supposed to be a duck,
then it should walk like a duck and quack like a duck!