Designing Modules In Python

Version 1.0
19th Feb 2017

Sripathi Krishnan, CTO, HashedIn Technologies

Table of Contents

Introduction

At HashedIn, we run a training program for junior developers to get better at design. During one of these programs, we asked developers to design a module to send sms. We also asked them to call the sms module from 3 different client applications. They had 30 minutes to complete this activity.

Then, after 30 minutes, we revealed a new set of requirements. Additionally, we set 2 hard rules for them to implement these requirements:

1. Developers were not allowed to change the way client applications invoked their sms module
2. Developers were not allowed to modify existing code. Instead, they had to write new code to address the additional requirements, without duplicating code.

20 minutes later, we again introduced a new set of requirements. By this time, most solutions broke down and failed to meet objectives in one way or other.

By gradually introducing additional requirements, and by forcing a set of constraints - developers were able to appreciate the need for a good design. They developed a mental model for discerning good design from bad.

We got the opportunity to review designs from 30+ developers, and learnt how developers think when asked to solve a problem. Eventually, we wrote a series of blogs with the ideal solution. The blogs tried to explain the thought process used to arrive at the solution, rather than simply presenting the solution.

This ebook is a more refined version of these blog posts. This is a part of our training material for HashedIn University, our developer bootcamp.

Who is this book for?

This book is for people who can program in an object oriented programming language. This book uses python to introduce various design patterns, but knowledge of python is not a must.

How to use this book

To make the most of this book, read the requirements at the start of each chapter, and then *write down* your solution. Then read through the rest of the chapter and critique your solution.

Good design is obvious once presented. But arriving at that solution is difficult, especially if you are new to programming. Writing down your solution is the only way to realize the shortcomings of your design. If you read the chapter without writing, you will tell yourself “That was obvious. I would have solved it in a similar manner”

Chapter 1: Designing Interface for SMS Client

Assume that you are the tech lead for Blipkart, an early stage ecommerce portal. Blipkart needs to send SMS (aka Text Messages) to customers at various stages:

- On checkout after payment is successful
- On failed checkout
- On successful delivery of products to consumer.

Since several business functionalities need to send SMS, you have been asked to design a reusable module. This module or function will be used by other developers (aka *clients* or *client developers*). You have to make it easy for them to send messages.

Blipkart has tied up with Watertel to send text messages. Watertel provides a HTTP POST based API to first login and then to send sms. Appendix A has details on the integration with Watertel.

For this exercise, you don't have to write complete code, pseudo code is sufficient. Concentrate on function signatures / class names, and think how client developers will use your module.

Also write the client code in orders.py and logistics.py to invoke the class or function that you have developed.

Tip	Before reading further, *write down* your solution on a piece of paper!

Step 1: Decide what's relevant to client developers

Your module sits between client developers on one side, and the API provided by Watertel on the other side. The module's job is to simplify sending an SMS, and to ensure that changes in Watertel do not affect your client developers.

A common way to simplify interaction with a third party service is to hide pieces of information that are not relevant to client developers.

Watertel's API exposes the following - username, password, access_token, expiry, phone number, message and priority. Our first job is to find out which of these concepts client developers care about.

Client developers don't care about username and password. They don't want to worry about expiry either. These are things our module should handle. They only care about phone number and message – "T*ake this message, and send it to this phone number*".

> Client developers only care about phone number and message – "take this message, and send it to this phone number".

Except phone number and message, all other parameters are implementation details. It is our module's responsibility to hide them from our clients.

Step 2: Define the Interface

Since developers only care about phone_number and message, our interface is very clear:

sms.py - with just a single function

```
# Client developers will call this function
# when they want to send a SMS
def send_sms(phone_number, message):
    pass
```

Now, let's look at it from our perspective - what else do we need to send the sms? We need the url, the username and password. We also need to manage the access token and handle its expiry. How will we get this data?

A naive approach is to read it from django settings, like this:

sms.py - Reading directly from settings [WRONG]

```
# Client developers will call this function
# when they want to send a SMS
```

```
def send_sms(phone_number, message):
    url = settings.sms_url
    username = settings.sms_username
    password = settings.sms_password

    # TODO: Now that we have everything, call Watertel's API
    # and send the SMS
```

This naive approach though has a major problem - it restricts you to exactly one configuration. Why would you want multiple configurations though?

Perhaps you want to unit test your code without actually connecting to Watertel. Or perhaps you have multiple accounts with Watertel, each with different usage limits. Or perhaps different modules want to send SMS using a different phone number.

When you read directly from settings, you are unnecessarily limiting your design.

In general, if your class needs some data, ask for it in your constructor. Make it somebody else's problem to pass you that information. It's not your job to look for that information.

> I *want* this piece of information to do my job. I don't care how you get it, but I need it before I can do anything.

Tip	This principle of asking what you want via your constructor is called *Dependency Injection*. Read this Stack Overflow question to learn more about the pattern.

Based on the above discussion, we first create a class - `SmsClient`. Then, we declare `url`, `username` and `password` as constructor arguments. It now becomes someone else's responsibility how these parameters are provided. We just assume we have these parameters and do our job of sending SMS.

sms.py - creating a SmsClient class

```
class SmsClient(object):
    def __init__(self, url, username, password):
        self.url = url
        self.username = username
        self.password = password

    # Client developers will call this function
    # when they want to send a SMS
    def send_sms(phone_number, message):
        # TODO - write code to send sms
        pass
```

Step 3: Invoke our class from client code

Next, in order to use our module, our clients need an object of `SmsClient`. To create an object though, they will need to provide url, username and password. Our first attempt could be something like this:

orders.py: Attempt #1 - using SmsClient class

```
# In orders.py
from django.conf import settings
from sms import SmsClient

# Create an object once
sms_client = SmsClient(settings.sms_url, settings.sms_username,
settings.sms_password)

# When ready, call the send_sms method
sms_client.send_sms(phone_number, message)
```

There are few problems with our first attempt:

1. First, `orders.py` shouldn't care how `SmsClient` objects are constructed. If we later need additional parameters in the constructor, we would have to change `orders.py`, and all classes that use `SmsClient`.
2. If we later decide to read our settings from some other place - say environment variables - then `orders.py` would have to be modified
3. We have to create `SmsClient` objects in every module that wants to send a SMS, which is way too much duplication

The solution is to create `SmsClient` object in sms.py module. Then orders.py and logistics.py can directly import the object from sms module.

Here is how it looks:

sms.py

```
from django.conf import settings

class SmsClient(object):
    def __init__(self, url, username, password):
        self.url = url
        self.username = username
        self.password = password

    def send_sms(phone_number, message):
        # TODO - write code to send sms
        pass

sms_client = SmsClient(settings.sms_url, settings.sms_username,
settings.sms_password)
```

orders.py:

```
from sms import sms_client
...
# when you need to send an sms
sms_client.send_sms(phone_number, message)
```

Tip	In this exercise, we are building a reusable module as part of a larger application. Therefore, it is okay for `sms.py` to construct the object and make `sms_client` available to other modules. However, if we were to make a truly reusable module that would installed via pip – then we shouldn't be constructing the object within sms.py. It'd be the client's responsibility to construct the object.

Summary

Before you start implementing, think how your clients will use your module. It’s a good idea to actually write down the client code (i.e. code in orders.py and logistics.py) before you start implementing your module.

Chapter 2: Implementing SmsClient

Earlier, we wrote a stub for our `SmsClient` class. In this chapter, we will build the logic to connect with Watertel.

If you read Watertel's developer documentation [see Appendix A], the steps to send an SMS are:

1. Login by making a post request to `url`, and pass a valid `username` and `password`.
2. If successful, you get back an `access_token` and an expiry timestamp. Store this `access_token` for future use.
3. When you want to send a SMS, make a post request that includes the `access_token`, the phone number and the message.

Step 1: Writing login method

We start by writing a `_login` method that stores the access token and it's expiry in member variables.

sms.py

```
class SmsClient(object):
    def __init__(self, url, username, password):
        self.url = url
        self.username = username
        self.password = password
        self.access_token = None
        self.expires_at = 0

    def _login(self):
        # TODO - Make HTTPS request, get accessToken and expiresAt
        # TODO - error handling, check status codes
        self.access_token = response["accessToken"]
        self.expires_at = get_current_time() + response["expiry"]
```

Notice that we prefix the function with an underscore. This denotes that the method is private. You don't want client developers to think about login, and hence we make the login method private.

Someone needs to call `_login`. Obviously, the clients shouldn't call `_login` directly. The next obvious choice is calling the `_login` method from our constructor. However, constructors must not do real work. If we make network calls in the constructor, it makes the object difficult to test, and difficult to reuse.

This leaves us with only one choice - call `_login` from the `send_sms` method. Here's how it looks in practice:

Calling _login before sending sms if access token has expired

```
class SmsClient(object):
    def _login(self):
        # TODO - Make HTTPS request, get accessToken and expiresAt
        # TODO - error handling, check status codes
        self.access_token = response["accessToken"]
        self.expires_at = get_current_time() + response["expiry"]

    def _get_access_token(self):
        if (get_current_time() > self.expires_at):
             self._login()
        return self.access_token

    def send_sms(self, phone_number, message):
        access_token = self._get_access_token()
        # TODO: use the access_token to actually send the SMS
```

Step 2: Write code to make HTTP Requests

At this point we can start writing code to send the SMS. We will assume a `make_request` function that returns the parsed JSON object and the http status code.

```
class SmsClient(object):
    def send_sms(self, phone_number, message):
        access_token = self._get_access_token()
        status_code, response = self._make_http_request(
            access_token, phone_number, message)
```

```
    def _make_request(self, access_token, phone_number, message):
        # TODO - use requests library to make a POST request
        return status_code, response
```

Step 3: Handling HTTP Status Codes

What should we do with the status_code? One option is to return the status_code and let our clients handle it. That's a bad idea though.

We don't want our clients to know how we are sending the SMS. This is important – if the clients know how you send the sms, you cannot change how your module works in the future. When we return the HTTP status code, we are telling them indirectly how our module works. This is a leaky abstraction, and you want to avoid it as much as possible (though you can't eliminate it completely).

We can't return the status code, but we still want our clients to do some error handling. Since our clients are calling a python method, they expect to receive errors the pythonic way – which is Python Exceptions.

When it comes to error handling, you need to be clear whose problem it is. It's either the client developers fault, or your modules fault. A problem with your module's dependency (i.e. Watertel server down) is also your module's fault – because the client developer doesn't know Watertel even exists.

We will create an exception class for each possible error –

- `BadInputError` – a base error for incorrect inputs
- `InvalidPhoneNumberError` – when the phone number is malformed or wrong
- `InvalidMessageError` – when the message is longer than 140 characters
- `SmsException` – when it is our modules fault and we cannot send the sms. This tells our clients that calling send_sms again is safe and may work.

With this, here is how our code looks like:

Updated sms.py with error handling

```
def _validate_phone_number(self, phone_number):
    if not phone_number:
        raise InvalidPhoneNumberError("Empty phone number")
    phone_number = phone_number.strip()
    if (len(phone_number) > 10):
        raise InvalidPhoneNumberError("Phone number too long")
    # TODO add more such checks

def _validate_message(self, message):
    if not message:
        raise InvalidMessageError("Empty message")
    if (len(message) > 140):
        raise InvalidMessageError("Message too long")

def send_sms(self, phone_number, message):
    self._validate_phone_number(phone_number)
    self._validate_message(message)

    access_token = self._get_access_token()
    status_code, response = _make_http_request(access_token,
phone_number, message)

    if (status_code == 400):
        # This is Watertel telling us the input is incorrect
        # If it is possible, we should read the error message
        # and try to convert it to a proper error
        # We will just raise the generic BadInputError
        raise BadInputError(response.error_message)

     elif (status_code in (300, 301, 302, 401, 403)):
         # These status codes indicate something is wrong
         # with our module's logic. Retrying won't help,
         # we will keep getting the same status code
         # 3xx is a redirect, indicate a wrong url
         # 401, 403 have got to do with the access_token being
wrong
         # We don't want our clients to retry, so we raise
RuntimeError
         raise RuntimeError(response.error_message)
```

```
      elif (status_code > 500):
         # This is a problem with Watertel
         # This is beyond our control, and perhaps retrying would
help
         # We indicate this by raising SmsException
         raise SmsException(response.error_message)
```

Step 4: Handling network exceptions

There's one more thing that is missing – handling any exceptions that are raised by `_make_request` function call. Assuming you are using the wonderful requests library, this is the list of exceptions that can be thrown.

You can categorize these exceptions into two categories – safe to retry v/s not safe to retry.

- Safe to retry? Wrap the exception in a `SMSException` and raise it.
- Not safe to retry? Just let the exception propagate.

In our case, `ConnectTimeout` is safe to retry. `ReadTimeout` indicates the request made it to the server, but the server did not respond timely. In such cases, you can't be sure if the SMS was sent or not. If it is critical the SMS be sent, you should retry. If your end users would be annoyed receiving multiple SMSes, then do not retry.

You can handle these exceptions inside the `_make_request` method.

sms.py : handling exceptions raised by requests library

```
def _make_http_request(access_token, phone_number, message):
    try:
        data = {"message": message, "phone": phone_number,
"priority": self.priority}
        url = "%s?accessToken=%s" % (self.url, access_token)
        r = requests.post(url, json=data)
        return (r.status_code, r.json())
    except ConnectTimeout:
        raise SmsException("Connection timeout trying to send
SMS")
```

That covers most of the implementation of our module. We won't get into the code that actually makes the HTTP request – you can use the requests library for that.

Summary

1. Don't leak inner workings of your module to your clients, otherwise you can never refactor or change your implementation.
2. Raise appropriate errors to inform your clients what went wrong.

Chapter 3: Adding Retry Logic

In previous chapters, we designed the interface for our SmsClient, then implemented the interface, and then wrote unit tests for the implementation.

The code has made it to production. About 50 different modules in the code are using our SMS module to send text messages.

We now have a feature request from the product managers:

> Customers are complaining they don't receive SMS'es at times. Watertel recommends resending the SMS if the status code is 5xx. Extend the sms module to support retries with exponential backoff. The first retry should be immediate, the next retry within 2s, and the third within 4s. If it continues to fail, give up and don't try further.

Of course, our product manager assumed we would take care of a few things, viz

1. We will implement this in all the 50 modules that are sending sms'es.
2. There would be no regression bugs due to this change.

Don't modify code that's already working

Let's start planning our change. There are two seemingly opposing views here -

1. We don't want client developers to change their code, not even a single character.
2. At the same time, the SmsClient class we wrote earlier works great – and we don't want to change code that is already working.

Tip	The open/closed principle tells us we shouldn't modify existing source code just to add new functionality. Instead, we should find ways to extend it without modifying the source code.

> In our case, this means we shouldn't touch the source code for SmsClient just to add new features.

Using Inheritance

This is tricky situation, but as with everything else in software, this problem can be solved with a little bit of indirection. We need something sitting in between our clients and our SmsClient class. That *something* can be a derived class of SmsClient.

sms.py : Using inheritance to implement new requirements

```
# This is the class we wrote earlier
# We are not allowed to change this class
class SmsClient(object):
    def send_sms(self, phone_number, message):
        ...

class SmsClientWithRetry(SmsClient):
    def __init__(self, url, username, password):
        super(SmsClient, self).__init__(url, username, password)

    def send_sms(self, phone_number, message):
        # TODO: Insert retry logic here
        super(SmsClientWithRetry, self).send_sms(phone_number,
message)
        # TODO: Insert retry logic here

# Earlier, client was an instance of SmsClient, like this
# client = SmsClient(username, password)
# We now change it to be an instance of SmsClientWithRetry
# As a result, our client developers doesn't have to change
# They are simply importing sms_client from sms.py

sms_client = SmsClientWithRetry(username, password)
```

If you notice, using inheritance, we got ourselves a way to add retry logic without modifying the existing code that is already known to work.

Tip	In general, we should prefer composition over inheritance. In the next post, we will see the limitations of inheritance and why composition is better.

When should we retry?

With that done, we can now work on adding the retry logic. We don't want to retry our client gave us an invalid phone number or a bad message – because it is a waste of resources. Even if we retried 100 times, it won't succeed. We also don't want to retry if there is a logic problem in our module's code – because our code cannot fix itself magically, and retry is unlikely to help.

In other words, we only want to retry if Watertel has a problem and we believe retrying may end up delivering the message.

If you revisit our original `SmsClient` implementation, you will now appreciate the way we designed our exceptions. We only want to retry when we get an `SmsException`. All other exception types, we simply let the client developers deal with it.

Adding Retry Loop with Exponential Backoff

Whenever you retry calling an external system, you should add in some delay between requests. There is no point trying immediately after a request failed - you are likely to get back the same error.

There are two common strategies - a random delay, and exponential backoff. Exponential backoff increases the delay between attempts in an exponential manner - 2s, 4s, 8s, 16s and so on.

SmsClient with exponential backoff

```
class SmsClientWithRetry(SmsClient):
    def __init__(self, url, username, password):
        super(SmsClientWithRetry, self).__init__(
                    url, username, password)
        # TODO: Parameterize num_attempts and backoff
        self.num_attempts = 3
        self.backoff = 2
```

```
    def send_sms(self, phone_number, message):
        attempts = 1
        retries = self.num_attempts
        delay = self.backoff
        while retries > 1:
            try:
                return super(SmsClientWithRetry, self).send_sms(
                        phone_number, message)
            except SmsException as e:
                print("Attempt #%s to send SMS failed. Retrying in
%s seconds" % (attempts, delay))
                time.sleep(delay)
                attempts += 1
                retries -= 1
                delay *= self.backoff

        # Last attempt to send the sms
        # If this still fails, we want SmsException to be raised
        return super(SmsClientWithRetry, self).send_sms(
                        phone_number, message)
```

Using a Retry Decorator

Retrying with exponential backoff is a very common approach when integrating with external systems. It doesn't make sense to write that logic every time - write it once, and reuse it.

The easiest way to handle retries in a generic manner is via a decorator.

Tip	Decorator is a design pattern that lets you add logic *before* and *after* a function is called. In our case, we want to call the send_sms function, and we want to add retry logic before and after send_sms function call.

We will not write the decorator function ourselves. There are several versions available online that solve this problem:

1. See SaltyCrane's implementation - http://www.saltycrane.com/blog/2009/11/trying-out-retry-decorator-python/

2. See retrying library on pip - https://pypi.python.org/pypi/retrying

Summary

1. The **open/closed principle** tells us not to modify our already working code just to add new features. It's okay to change code for bug fixes, but other than that, we should look at functional / object oriented practices to extend the existing code when we want to add new functionality.
2. **Exponential backoff strategy** prevents overloading external system by increasing the delay between re-attempts
3. **Inheritance** can be used to extend logic without modifying code
4. **Decorators** help you write code before and after a function without actually modifying the function

Chapter 4: Integrating with a Second SMS Gateway

Previously, we added a retry mechanism to send sms. With that change, things are now stable in production.

Your finance team now wants a change. Watertel SMS gateway is proving out too expensive. A competitor - Milio - is now offering the same service at a better pricing.

Your product manager wants you to minimize the risk by routing only 20% of the messages through Milio. If Milio proves stable after a few weeks of usage, the product manager would slowly increase the traffic.

Of course, like last time round – we do not want to change existing code that is already working. Also, we need to have the retry logic for Milio as well. And finally, we do not want to duplicate any code.

Create an implementation for Milio

Milio's interface for sending SMS is different than Watertel. It doesn't have a username/password, and no API to login.

Instead, Milio provides an access token and a registered phone number. We need to add the access token as a HTTP request header. The registered phone number must be sent in the POST body.

Our first step is to create a new class `MilioSmsClient`, with the same interface as our original SmsClient. We have covered this before, so we will just write the pseudo-code below.

sms.py : Adding stub interface for MilioSmsClient

```
# This is the class we wrote earlier
# We are not allowed to change this class
class SmsClient(object):
    def send_sms(self, phone_number, message):
        ...
```

```
# We wrote this class to handle retries
# Again, we are not allowed to change this class
class SmsClientWithRetry(SmsClient):
    def send_sms(self, phone_number, message):
        ...

# This is our new class to send sms using Milio's APi
class MilioSmsClient(object):
    def __init__(self, url, registered_phone, access_token):
        self.url = url
        self.registered_phone = registered_phone
        self.access_token = access_token

    def send_sms(self, phone_number, message):
        # Similar to send_sms method in SmsClient
        # Difference would be in the implementation
        # We will have different JSON response,
        # and different request parameters
        print("Milio Client: Sending Message '%s' to Phone %s"
                % (message, phone_number))

# Client developers only care about sms_client object
# When ready, we will change sms_client to use our new class
sms_client = SmsClientWithRetry(username, password)
```

The next step is to actually write the logic in `send_sms` method. We will leave that as an exercise, as the logic is largely similar to what we did for Watertel.

Implementing retries without code duplication

`MilioSmsClient` needs to support retries, just like we did for our Watertel implementation. But we don't want to duplicate the code we wrote earlier. How can we achieve that?

There is a conflict situation:

1. Adding retries without code duplication would mean we have to inherit `SmsClientWithRetry`.
2. But `SmsClientWithRetry` assumes we need a url, username and password as constructor arguments - and we don't have that for Milio. Hence, inheritance is cumbersome

This conflict is largely because we made a mistake in the earlier chapter - we chose to build new logic using inheritance. Inheritance has tied us into a rigid hierarchy. We need to break this hierarchy.
To get around this problem, we will refactor our code. This time round, we will use *Composition* instead of Inheritance.

sms.py : Refactoring our code

```
# We rename SmsClient to WatertelSmsClient
# This makes the intent of the class clear
# We don't change anything else in the class
class WatertelSmsClient(object):
    def send_sms(self, phone_number, message):
        ...

# This is our new class to send sms using Milio's APi
class MilioSmsClient(object):
    def send_sms(self, phone_number, message):
        ...

class SmsClientWithRetry(object):
    def __init__(self, sms_client):
        self.delegate = sms_client

    def send_sms(self, phone_number, message):
        # Insert start of retry loop
        self.delegate.send_sms(phone_number, message)
        # Insert end of retry loop

_watertel_client = SmsClient(url, username, password)

# Here, we are pass watertel_client
# But we could pass an object of MilioClient,
# and that would still give us retry behaviour
sms_client = SmsClientWithRetry(_watertel_client)
```

Let's go through the changes:

1. **Renaming SmsClient:** We renamed `SmsClient` to `WatertelSmsClient`. When we started, SmsClient was descriptive

enough. Now with two different backend systems, SmsClient is confusing.

2. **No Inheritance**: `SmsClientWithRetry` gets a constructor argument called sms_client. When it wants to send a sms, it *delegates* the job to the sms_client that it got in the constructor
3. **Constructing sms_client**: Client developers only care about the `sms_client` object that they import. So, we smartly reconfigure sms_client in a way that provides the exact same functionality as before.

With *Inheritance*, we reused functionality by calling super.

```
super(SmsClientWithRetry, self).send_sms(phone_number, message)
```

With *Composition*, we reuse functionality by delegating to an object that was provided in the constructor.

```
self.delegate.send_sms(phone_number, message)
```

The advantage with Composition is that we can pass *any* object that has a `send_sms` method defined. In other words, we can pass either `WatertelSmsClient` or MilioSmsClient. That is powerful.

Splitting Traffic between Milio and Watertel

So far, we have a working implementation of MilioSmsClient, and we have a way to retry sending SMS that works with both our implementations. But how do we split the traffic between the two?

The actual logic to split traffic is simple. Generate a random number between 1 and 100. If it is between 1 and 80, use `WatertelSmsClient`. If it is between 81 and 100, use `MilioSmsClient`. Since the number is random, over time we will get a 80/20 split between the implementations.

But, the big question is – where should we put this logic?

As before, we don't want client developers to implement this logic. Hell, we don't even want them to know we are using 2 different providers. Additionally, we can't add that logic to either `WatertelSmsClient` or

`MilioSmsClient` - those classes are already working, and we don't want to touch code that is already working well.

The only way out is one more indirection. We create an intermediate class that is used by all client developers. This class will decide how to split traffic between `MilioSmsClient` and `WatertelSmsClient`. Let's call this class `SmsRouter`.

sms.py : Adding a SmsRouter to split traffic

```
from random import randint

class WatertelSmsClient(object):
    ...

class MilioSmsClient(object):
    ...

class SmsClientWithRetry(object):
    ...

class SmsRouter:
    def __init__(self, split_ratio, watertel, milio):
        # TODO: assert split_ratio is between 1 and 100
        self.split_ratio = ratio
        self.watertel = watertel
        self.milio = milio

    def send_sms(self, phone_number, message):
        number = randint(1, 100)
        if number <= self.split_ratio:
            self.watertel.send_sms(phone_number, message)
        elif number <= 100:
            self.milio.send_sms(phone_number, message)
        else:
            raise SmsException("Unreachable code path!")

_watertel = WatertelSmsClient(url, username, password)
_milio = MilioSmsClient(url, registered_phone, access_token)
sms_client = SmsRouter(80, _watertel, _milio)
```

There are two things to note about this implementation -

1. `SmsRouter` has the exact same signature for the `send_sms` method. This is important - otherwise client developers would have to change their code.
2. `SmsRouter` doesn't actually do the work of sending the sms. It only decides who should send the sms, and then it simply delegates it to the appropriate object.

There is one problem though with `SmsRouter`. It either delegates to milio or watertel. But, the router shouldn't care to whom it is delegating - it could be a totally different implementation for all it cares.

We just need to rename our variables to be a bit more generic.

sms.py : Version 2 of SmsRouter without dependency on milio or watertel

```
from random import randint

class WatertelSmsClient(object):
    ...

class MilioSmsClient(object):
    ...

class SmsClientWithRetry(object):
    ...

class SmsRouter:
    def __init__(self, split_ratio, first, second):
        # TODO: assert split_ratio is between 1 and 100
        self.split_ratio = ratio
        self.first = first
        self.second = second

    def send_sms(self, phone_number, message):
        number = randint(1, 100)
        if number <= self.split_ratio:
            self.first.send_sms(phone_number, message)
        elif number <= 100:
            self.second.send_sms(phone_number, message)
        else:
```

```
            raise SmsException("Unreachable code path!")

_watertel = WatertelSmsClient(url, username, password)
_milio = MilioSmsClient(url, registered_phone, access_token)
sms_client = SmsRouter(80, _watertel, _milio)
```

Now we can pass any two implementations we want, and `SmsRouter` will split the traffic appropriately between the two.

Tip	You can go a step further and modify `SmsRouter` to accept any number of implementations instead of exactly two. But, you should practice YAGNI - you ain't gonna need it. There is no point generalizing beyond your current needs. YAGNI is a conscious effort. In the first version, even I wrote the more general version. It's easy to fall into the trap of generalizing more than necessary - always be on the watch out. I have left the general version below, but remember - YAGNI.

Combining Retries and SmsRouter

We have one last thing to do - combine `SmsClientWithRetry` and and `SmsRouter`. This is now easy to do - it's just a matter of constructing objects in the right order.

sms.py : Combining retries and split traffic

```
from random import randint

class WatertelSmsClient(object):
    ...

class MilioSmsClient(object):
    ...

class SmsClientWithRetry(object):
```

```
    ...

class SmsRouter:
    ...

# First, create implementation specific objects
_watertel = WatertelSmsClient(url, username, password)
_milio = MilioSmsClient(url, registered_phone, access_token)

# Then, create the router to split traffic
_router = SmsRouter(80, _watertel, _milio)

# Finally, wrap the router to allow for retries
# This final object will be used by client developers

sms_client = SmsClientWithRetry(_router)
```

One nice thing about this is how retries now work. If one provider fails, retry may end up sending the sms using another provider. In other words, retries are across providers - you don't keep retrying with the same provider.

Summary

1. **Prefer Composition over Inheritance** - Composition provides a better way to reuse code
2. **Don't be afraid to refactor** - You can't get the design right the first time. Refactor when necessary.
3. **YAGNI** - Don't over generalize code. Solve your specific problem, you can always refactor later if necessary.

Appendix A: Watertel Integration Guide

Once you buy a plan from Watertel, you will be provided with UserName and Password. There are two APIs that you will need to call - login and send sms.

Login API

Before you send an SMS, you need to login and get an access token. The access token is valid for a few hours, and you don't have to login again as long as the access token is valid.

Make a POST request to https://watertel.com/api/login.
Content type must be application/json.

Request Body

```
{
    "username" : "testuser",
    "password" : "password"
}
```

Response Body

```
{
    "accessToken": "jdsji423kjkjskufiajk32j324",
    "expiresIn" : 7200
}
```

Send SMS API

To send an SMS, make a POST request to
https://watertel.com/api/sms?accessToken=<>

Request Body

```
{
    "message": "Your order was successful",
    "phone" : "9986650980",
    "priority': 1
}
```

Response Status Code	**Meaning**
200	SMS delivered successfully
400	Invalid arguments. Either wrong phone number, or the text was longer than 140 characters
401	Invalid access token
500	Problem with watertel gateway. SMS was not delivered. Application can retry the request.

Appendix B: Milio Integration Guide

Once you create an account with Milio, you will be given the following -

1. Access token
2. Registered phone number

Any Sms you send will appear as though it came from your registered phone number.

To send a SMS, send a POST request to https://milio-sms-gateway.com/api/sms?accessToken=<>

Request Body

```
{
   “text”: “Hello, SMS!”,
   “recipientPhone”: “9986650980”,
   “registeredPhone”: “9342414106”
}
```

Response Status Code	Meaning
201	SMS delivered successfully
400	Invalid arguments. Either wrong phone number, or the text was longer than 140 characters
401	Invalid access token
429	Indicates you are trying to send too many messages and have been rate limited.
500	Indicates an internal error, you can retry