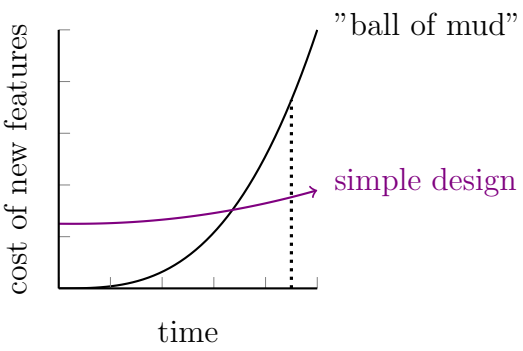
CO220 - Software Engineering Design

8th October 2019

Cost of Change



The "project heat death", denoted by the dotted line, is where the cost of adding new features outweighs the value gained by adding those features. Note that the initial cost of doing a simple design can be more expensive (since it requires more planning).

Elements of Simple Design

This is arranged in a pyramid on the slides (since they "build up on each other") but I will write it as a list, starting from the bottom;

1. **behaves correctly**

   It doesn't matter if the codebase is well structured, or the code is elegant if it doesn't do the right thing (is buggy, or isn't what the customer wanted).

   - automated testing
   - test-driven development
   - mock objects

2. **minimises duplication**

   If something needs to be changed in the future, and it's in multiple places, it will have to be changed in all of those places which will take longer. Additionally, it's also easy to miss, causing bugs.

3. **maximises clarity**

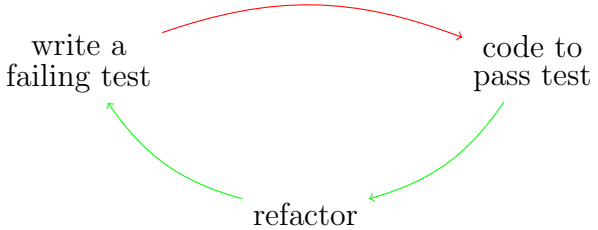
   Code should be easy to modify, such that the parts that need to be changed can be easily located. Important especially if working with others.

4. **has fewer elements**

   Less important - we want to focus on the previous levels first, and don't want to lose the benefits by combining elements.

Test-Driven Development / Behaviour Driven Development

Having a test suite provides confidence that the codebase still works, even after major changes.



We start by writing failing tests, which seems counterintuitive, as there is no code to test. However, these tests are written as if the code was working - which gives us a specification on how the code **should** behave. We want to write code as quickly as possible that gets us from the red state (failing tests), to a green state (passing tests). This code is likely untidy - we can then tidy it up (which shouldn't break the tests).

Additionally, it's not only about testing; we can replace the stages as follows;

- API design (write a failing test)

  "I wish there was a method that would take these parameters and do this"

- internals design (code to pass test)

  "Just making it work"

- structural design (refactor)

  "How can we improve the design?" (pyramid layers)

We focus more on what it should do (how it should behave), and not how it does it. For example, `CustomerLookup` should do the following;

- finds customer by ID
- fails for duplicate customers

The test is named as the expected result, and if it is true, then it behaves correctly.

```
public class CustomerLookupTest {
  @Test
  findsCustomerById() {
    ...
  }
  @Test
  failsForDuplicateCustomers() {
    ...
  }
}
```

Example of TDD

The object `FibonacciSequence` should do the following;

- defines the first two terms to be one
- has each term equal to the sum of the previous two
- is not defined for negative terms

```
... // (FibonacciSequenceTest.java)

import static org.hamcrst.CoreMatchers.is;
import static org,junit.Assert.assertThat;

import org.junit.Test;

public class FibonacciSequenceTest {

  @Test
  public void definesFirstTwoTermsToBeOne() {
```

```
12      assertThat(new FibonacciSeqeunce().term(0), is(1));
13      assertThat(new FibonacciSeqeunce().term(1), is(1));
14    }
15  }
```

Obviously, none of this will work yet, as the code doesn't exist. However, we can use this to create the code as follows (this is incorrect, but our tests now pass);

```
1  ... // (FibonacciSequence.java)
2
3  public class FibonacciSequence {
4
5    public int term(int i) {
6      return 1;
7    }
8  }
```

We can then add more tests, which should now fail;

```
1  ... // (FibonacciSequenceTest.java)
2
3  public class FibonacciSequenceTest {
4    ...
5
6    @Test
7    public void hasEachTermTheSumOfPreviousTwo() {
8      assert(new FibonacciSeqeunce().term(2), is(2));
9      assert(new FibonacciSeqeunce().term(3), is(3));
10     assert(new FibonacciSeqeunce().term(4), is(5));
11   }
12 }
```

Similarly, we can modify the code again to add a naive implementation which performs it recursively;

```
1  ... // (FibonacciSequence.java)
2
3  public class FibonacciSequence {
4
5    public int term(int i) {
6      if (i < 2) {
7        return 1;
8      }
9      return term(i - 1) + term(i - 2);
10   }
11 }
```

Adding the last bullet point as a test;

```
1  ... // (FibonacciSequenceTest.java)
2
3  public class FibonacciSequenceTest {
4    ...
5
6    @Test
7    public void isNotDefinedForNegativeIndices() {
8      try {
9        new FibonacciSeqeunce().term(-1);
10       fail("should have thrown exception")
```

```
    } catch (IllegalArgumentException iae) {
      assertThat(iae.getMessage(), containsString("negative index"))
    }
  }
}
```

Fixing this, we add the following;

```
... // (FibonacciSequence.java)

public class FibonacciSequence {

  public int term(int i) {
    if (i < 0) {
      throw new IllegalArgumentException("negative index not supported");
    }

    ...
  }
}
```

This is the only time I will actually write out every step, since that's the focus of TDD.

11th October 2019

Feedback

Note that the names of the test files should be `SomeObjectTest`, for `SomeObject`. This convention allows the IDE to link the files, as well as having them in alphabetical order. Also always use a *jUnit* library function;

`assertThat(rul.size(), is(0))` or `assertEquals(0, rul.size())`
instead of
`assert rul.size() == 0`

Generally make the LHS of fields the interface `List` instead of `ArrayList`, and attempt to make it `private` and `final` (if possible). Additionally, any fields are reinitialised automatically by *jUnit*, hence it doesn't need to be reset at the end of each test.

Refactoring

This starts with multiple examples on handouts. As we're writing new code, we should look out for small changes that can improve the structure of our code.

We can accumulate "technical debt" by writing code quickly to get a feature working, but we must fix it soon, otherwise it builds up leading to unhygienic code.

Note that refactoring should be done with tools when possible (such as renaming identifiers), since the tool will be able to analyse the entire codebase to detect where changes need to be made. Behaviour should not be changed.

The example after this is mostly using *IntelliJ* tools wherever possible. One note to make is that sometimes it is helpful to get code into a state where it becomes similar enough to other parts of the code, to allow for the tool to do the work.

15th October 2019

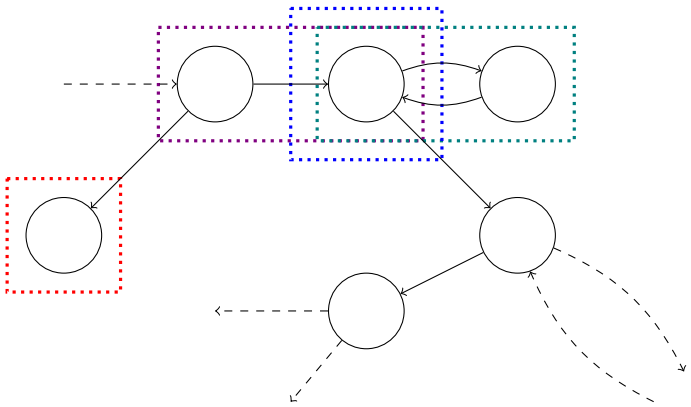
Sending Messages

Instead of considering how objects call methods, it may be beneficial to design how modules communicate with each other ("send messages").

The idea is that when one object "sends a message", we don't really care how it does it, just that it performs the expected action.

OOP

Typically, larger systems are built up of smaller units that work together. Some of these will be from the standard library, some of those will be written by us, and some others may be written by third parties.



However, these components should be reusable, and they can be combined in a different way if needed (we can't really modify code in the standard library, etc). We want to have the possibility to swap out parts of the project without affecting other objects. The system shouldn't care how the object does a job, just that it does it.

- **commands** "please do X"

  This doesn't wait for a response, or a even a return value.

- **queries** "please tell me Y"

  Requests a bit of data, and then processes it. If too many queries are used, we tend to have a central part of the process that deals with all the information which isn't flexible. Ideally jobs are delegated to different components.

- **value objects**

  These don't typically interact with other objects, and just holds some data and performs some computations. These components can easily be tested.

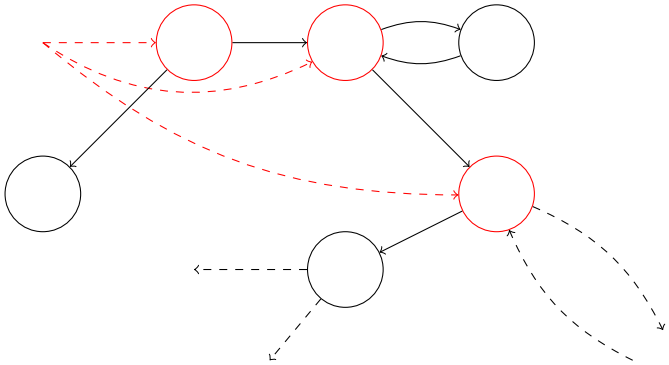
- **tell don't ask**

  The role of this object is to coordinate other objects in the system. Focusing on commands tends to give us more flexibility, but leads to different testing approaches.

Typically, asking looks like the following (can be characterised by a higher number of `getX()`s);

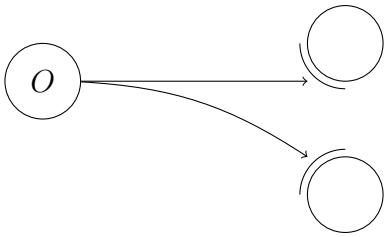
```
1 table.getGrid().getColumnModel().getColumn(index).setPreferredWidth(newWidth);
```

Visually, the graph becomes something similar to this;



Testing Objects and Roles

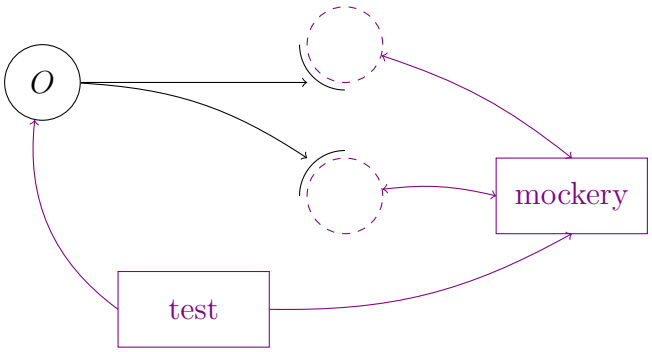
When we want to test a given object, let it be O, we shouldn't have to implement the objects it depends on first. To do this, we use interfaces in Java to represent roles - which specifies what O expects the other objects to do. An object may play different roles (hence implement multiple interfaces).



The object O is the one we wish to test, and should be triggered by a call in the test suite.

Mock Objects

To test objects that rely on stuff other objects that haven't been implemented yet (only an interface exists), we can use mock objects.



When a test is run, we can test if O sends messages to the mock objects. Instead of writing assertions about the state of O, we can write expectations for what should be called.

An example is as follows (this is syntax from *WebSequenceDiagrams*, because I don't want to draw it out);

```
Waiter->HeadChef: order(CHICKEN, APPLE)
HeadChef->PastryChef: order(APPLE)

Waiter->HeadChef: customerReadyFor(APPLE)
HeadChef->PastryChef: isCooked(APPLE)
PastryChef-->HeadChef: True
HeadChef->Waiter: serve(APPLE)

Waiter->HeadChef: customerReadyFor(APPLE)
HeadChef->PastryChef: isCooked(APPLE)
PastryChef-->HeadChef: False
```

In Java, we have the following for the tests;

```
... // (HeadChefTest.java)

public class HeadChefTest {

  @Rule
  public JUnitRuleMockery context = new JUnitRuleMockery();

  public final Order APPLE_TART = new Order("apple");
  public final Order ROAST_CHICKEN = new Order("chicken");

  Chef pastryChef = context.mock(Chef.class);
  RestaurantWaiter waiter = context.mock(RestaurantWaiter.class);

  @Test
  public void delegatesDessertToPastryChef() {

    HeadChef headChef = new HeadChef(pastryChef, waiter);

    context.checking(new Expectations() {{
      exactly(1).of(pastryChef).order(APPLE_TART);
    }});

    headChef.order(ROAST_CHICKEN, APPLE_TART);
  }

  @Test
  public void asksWaiterToServeDessertIfCooked() {

    HeadChef headChef = new HeadChef(pastryChef, waiter);

    context.checking(new Expectations() {{
      exactly(1).of(pastryChef).isCooked(APPLE_TART); will(returnValue(true));
      exactly(1).of(waiter).serve(APPLE_TART);
    }});

    headChef.customerReadyFor(APPLE_TART);
  }

  @Test
  public void doesNotAskWaiterToServeDessertIfNotCooked() {

    HeadChef headChef = new HeadChef(pastryChef, waiter);

    context.checking(new Expectations() {{
      exactly(1).of(pastryChef).isCooked(APPLE_TART); will(returnValue(false));
      never(waiter).serve(APPLE_TART);
    }});

    headChef.customerReadyFor(APPLE_TART);
  }
}
```

Similarly for the `HeadChef`;

```
... // (HeadChef.java)

public class HeadChef {

  private final Chef pastryChef;
  private final RestaurantWaiter waiter;

  public HeadChef(Chef pastryChef, RestaurantWaiter waiter) {
    this.pastryChef = pastryChef;
    this.waiter = waiter;
  }

  public void order(Order main, Order dessert) {
    pastryChef.order(dessert);
  }

  public void customerReadyFor(Order dessert) {
    if (pastryChef.isCooked(dessert)) {
      waiter.serve(dessert);
    }
  }
}
```

Note that we have **interfaces** for `Chef` and `RestaurantWaiter`, as they aren't implemented;

```
... // (Chef.java)

public interface Chef {
  void order(Order order);

  bool isCooked(Order order);
}
```

```
... // (RestaurantWaiter.java)

public interface RestaurantWaiter {
    void serve(Order order);
}
```

18th October 2019

Feedback

We only want to test each behaviour once, to avoid breaking tests elsewhere. Note that *jMock* is more strict than *Mockito*. Additionally, if we are expecting the same value in the behaviour, we don't have to make a new mock object if it won't be tested (hence we can just create a constant field).

```
... // (CameraTest.java)

public class CameraTest {
  ...

  private static final byte[] PHOTO = new byte[8];
  ...
```

```
8
9     @Test
10    public void switchingTheCameraOffPowersDownTheSensor() {
11
12      context.checking(new Expectations() {{
13        ignoring(sensor).powerUp();
14        exactly(1).of(sensor).powerDown();
15      }});
16
17      camera.powerOn();
18      camera.powerOff();
19    }
20
21    @Test
22    public void pressingTheShutterCopiesData() {
23
24      context.checking(new Expectations() {{
25        ignoring(sensor).powerUp();
26        exactly(1).of(sensor).readData(); will(returnValue(PHOTO));
27        exactly(1).of(memoryCard).write(PHOTO);
28      }});
29
30      camera.powerOn();
31      camera.pressShutter();
32    }
33  }
```

Designing for Flexibility

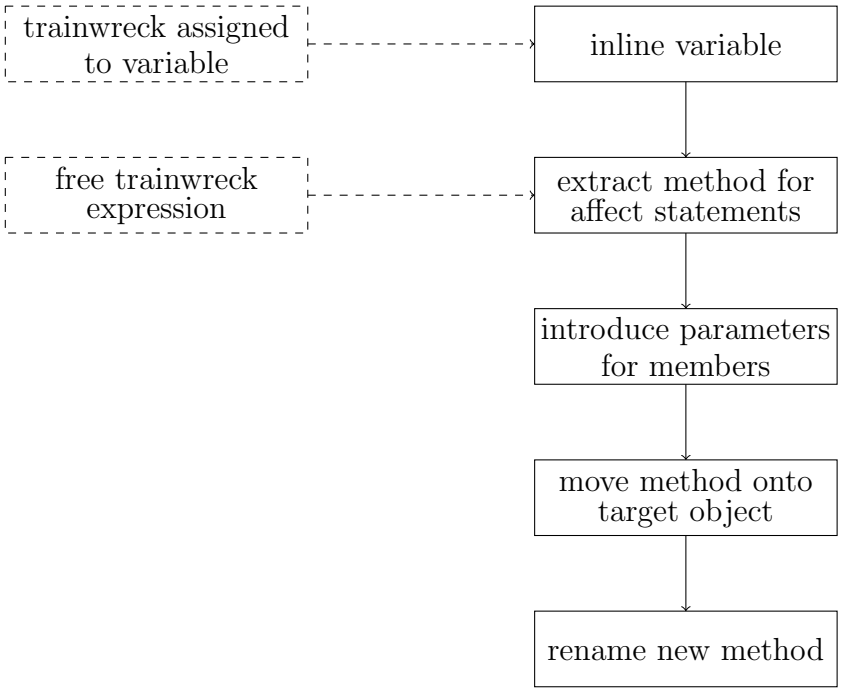
Bad design consists of the following properties;

- **rigidity** difficult to change, possibly due to complex code (long methods, deep conditions etc.)
- **fragility** making a change in one place could break another part of the code
- **immobility** difficult to reuse code in another context

Law of Demeter

Recall the graph that had many `getX()`s, the red lines reached across the graph, but the law of Demeter states that access should generally be to an object that is one "hop" away. This preserves flexibility. Violating this can cause fragility, as changing one part of the code could affect another part of the code that is far away.

We can perform the following steps to extract a "trainwreck" into a method, which is then put into the next object down the "chain" of `getX()`s.



Defending against Null Pointer Exceptions

The following snippet of code has lines 3 and 5 added to protect against NPEs - however if this is needed frequently it could lead to code duplication;

```
1 void playTrack(String name) {
2   Track track = library.getTrack(name);
3   if (track != null) {
4     track.play();
5   }
6 }
```

Another approach is to have the **null object pattern**, which is an empty implementation;

```
1 interface Track {
2   public void play();
3 }
4
5 class NullTrack implements Track {
6   public void play() {
7     // do nothing
8   }
9 }
```

As a development team, it makes sense to agree on what will be done, whether it be using the null object pattern, or using Java's `Optionals`.

Coupling and Cohesion

- aim for **low coupling** between classes — changing one part requires a change in the other
- aim for **high cohesion** within each class — a class should be specialised (less changes needed)

Ideally, we want to limit the "blast radius" of our changes, which is the amount of code we affect to just parts managed by us.

Approaches

One extreme is to store all of the code in a single repository, allowing changes to be made when needed (given approval), which is done by Google. This has the benefit that a part can be changed in part of the code, and can also be fixed in another. However, due to the ability to affect other unrelated parts of the codebase, it can also lead to issues when updating a core object.

The other extreme is to have modular code, which is individually versioned. That way, if something is updated, other modules can use older versions and update when needed (which doesn't break functionality straight away). However, updates will need to be done quite frequently, otherwise other modules will be behind. It's also difficult to make changes in other parts.

22nd October 2019

Motivation

This focuses mostly on the middle two layers in the pyramid mentioned in the first lecture. We assume that the code is working correctly, and we can check that the code is working correctly with the test suite.

Example

`ReverseEncoder` is an encoder for encryption, which reverses each word in the input (such that `"abc 123"` becomes `"cba 321"`). This will be done in Python;

```
1 class ReverseEncoder:
2   def encode(self, line):
3     words = line.split(" ")
4     results = []
5     for word in words:
6       results.append(word[::-1])
7     return " ".join(results)
```

Now suppose we have another encoder, `DoublingEncoder`, that repeats each word in the input (such that `"abc 123"` becomes `"abcabc 123123"`). This can easily be implemented as follows;

```
1 class DoublingEncoder:
2   def encode(self, line):
3     words = line.split(" ")
4     results = []
5     for word in words:
6       results.append(word + word)
7     return " ".join(results)
```

However, notice that there is a significant amount of duplication, as essentially all lines are the same, other than line 6. To remedy this, we can add an `Encoder` class which they both extend;

```
1  from abc import abstractmethod
2
3  class Encoder:
4    def encode(self, line):
5      words = line.split(" ")
6      results = []
7      for word in words:
8        results.append(self.encode_word(word))
9      return " ".join(results)
10
11   @abstractmethod
```

```
12    def encode_word(self, word):
13      pass
14
15  class ReverseEncoder(Encoder):
16    def encode_word(self, word):
17      return word[::-1]
18
19  class DoublingEncoder(Encoder):
20    def encode_word(self, word):
21      return word + word
```

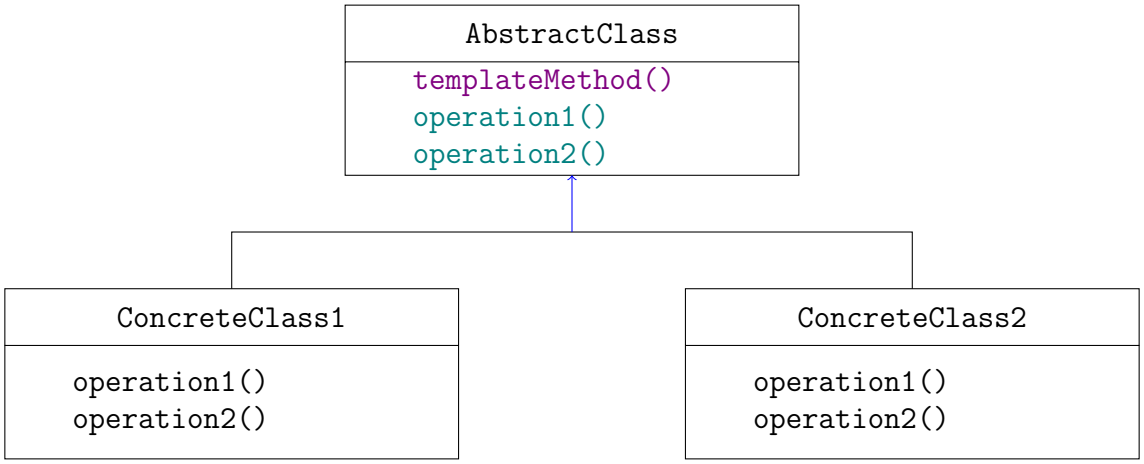
Now, everything that is common to the two algorithms is in the superclass, and only the part that is specialised is each individual class.

Template Method Pattern

The template method pattern is used when requirements change over time, and we need to vary a small part of an algorithm. A skeleton should be defined (our `Encoder` in the example above), and the vary steps should be deferred to subclasses (`ReverseEncoder`, and `DoublingEncoder`).

Separate things that change from things that stay the same.

Generally, this has the following pattern, where the specialised hook methods are overwritten by the inheriting classes, and the template method contains the shared behaviour;



This follows the **open-closed principle**, where we can extend the class' behaviour without modifying it, as we can just add a new subclass. We therefore don't have to edit code, which may have lead to bugs, as we are safer by just adding code. Modules should be open for extension, but closed to modification. Change behaviour by adding new code, not by changing existing code.

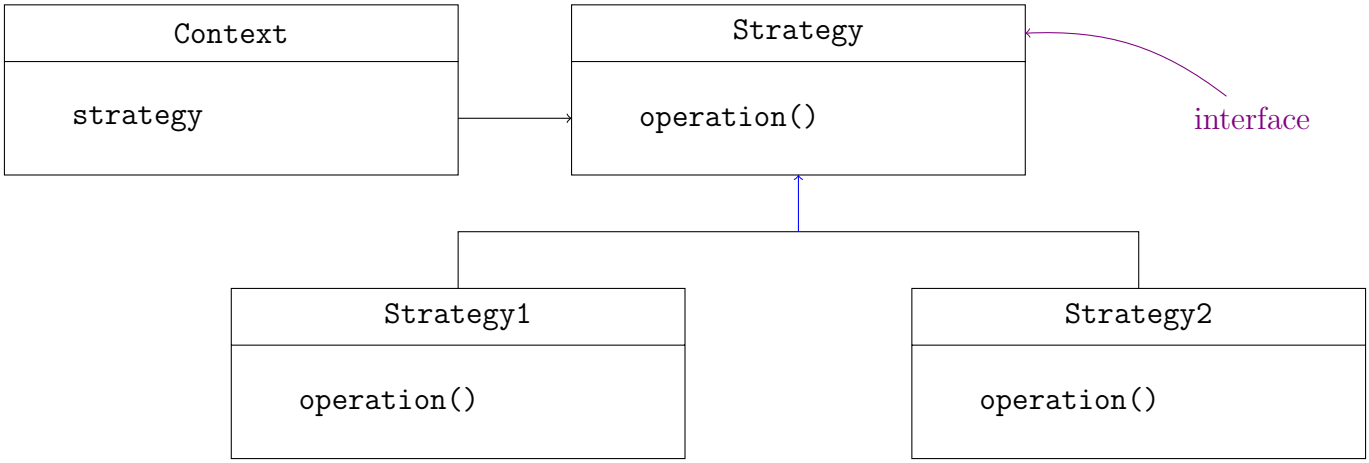
Coupling Metrics

Afferent coupling (Ca) - remember this as how many arrows **a**rrive - a class' afferent couplings is a measure of it's responsibility (how many other classes use it). Efferent coupling (Ce) - remember this as how many arrows **e**xit - it's a measure of how many different classes are used by this specific class (measures independence).

For example, the `ReverseEncoder` cannot really be used elsewhere, as it has a strong coupling with the `Encoder` superclass. In general, we may want to avoid this coupling.

Strategy Pattern

An alternative is the template method (still done for the same reason) is to delegate to a collaborator (instead of a subclass) - the algorithm should be pulled into a separate object. This favours object composition over class inheritance.



It's important to note that `Strategy` is an interface, hence `Strategy1` and `Strategy2` implement it. The example can be modified as follows;

```
from abc import abstractmethod

class Encoder:
  def __init__(self, encryptor):
    self.encryptor = encryptor

  def encode(self, line):
    words = line.split(" ")
    results = []
    for word in words:
      results.append(self.encryptor.encode_word(word))
    return " ".join(results)

class Reverser:
  def encode_word(self, word):
    return word[::-1]

class Doubler:
  def encode_word(self, word):
    return word + word
```

The main sign is that the strategy is passed in to the context; hence we would have;

```
Encoder(Reverser()).encode("...")
```

As the `Reverser` and `Doubler` no longer have a relationship with `Encoder`, we have must less coupling, hence we have better mobility and flexibility. Generally, composition should be preferred over inheritance.