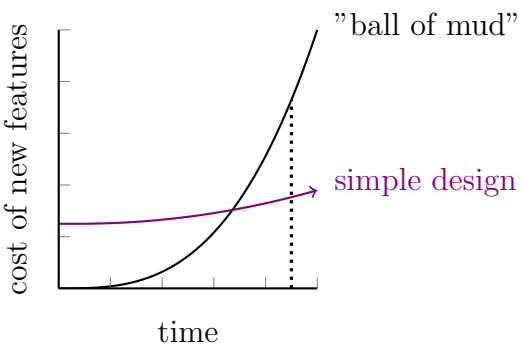
CO220 - Software Engineering Design

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**

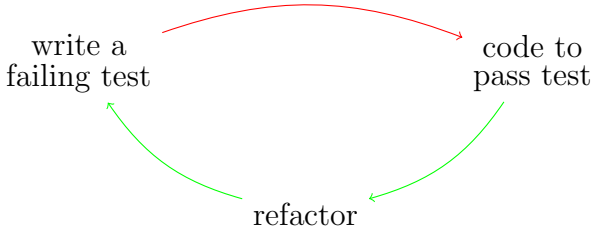
   Code should be easy to modify, such that the parts that need to be changed can be easily located. Important especially if working with others.

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

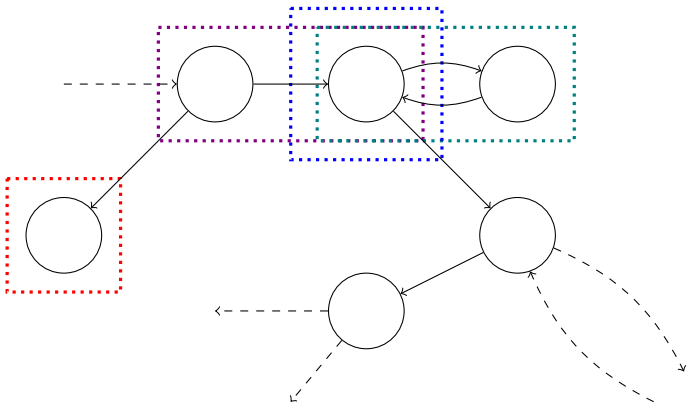
Sending Messages

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.

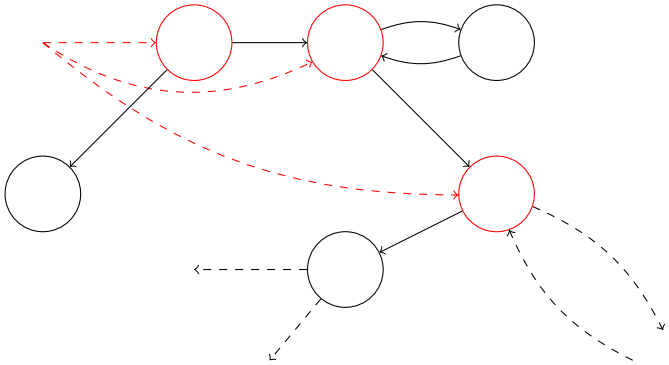
- **tell don't ask**

  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);

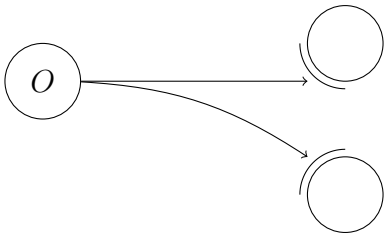
```
1 table.getGrid().getColumnModel().getColumn(index).setPreferredWidth(newWidth);
```

Visually, the graph becomes something similar to this;



Testing Objects and Roles

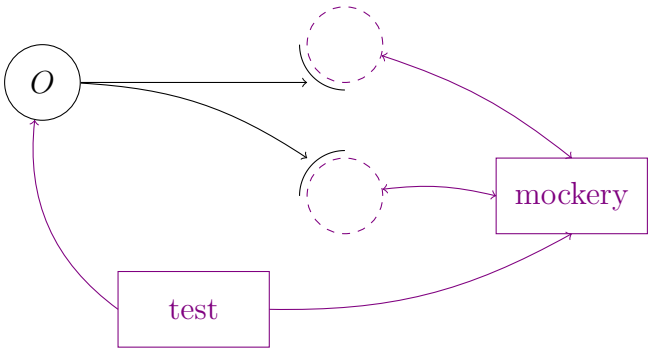
When we want to test a given object, let it be O, we shouldn't have to implement the objects it depends on first. To do this, we use interfaces in Java to represent roles - which specifies what O expects the other objects to do. An object may play different roles (hence implement multiple interfaces).



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

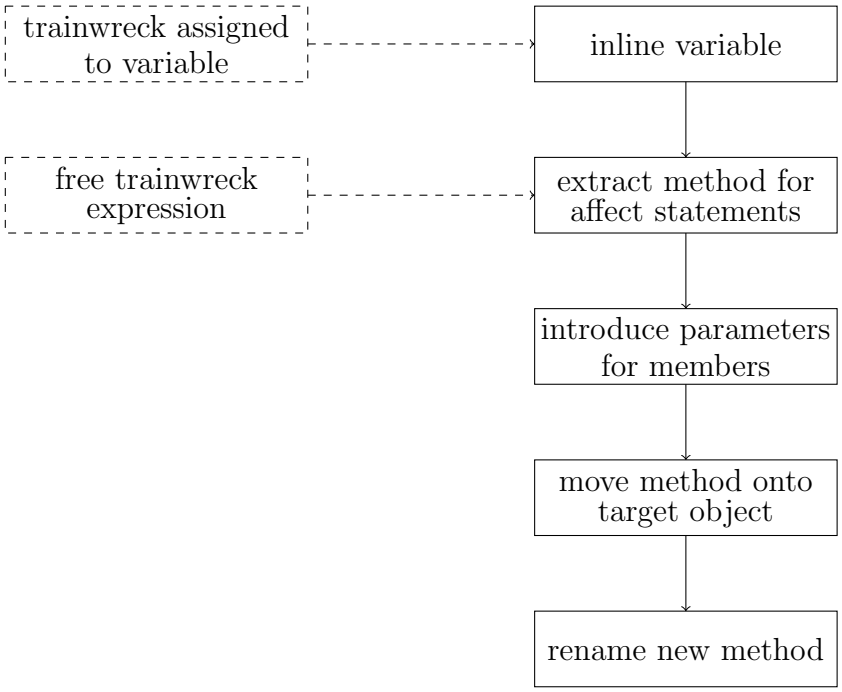
Bad design consists of the following properties;

- **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
```

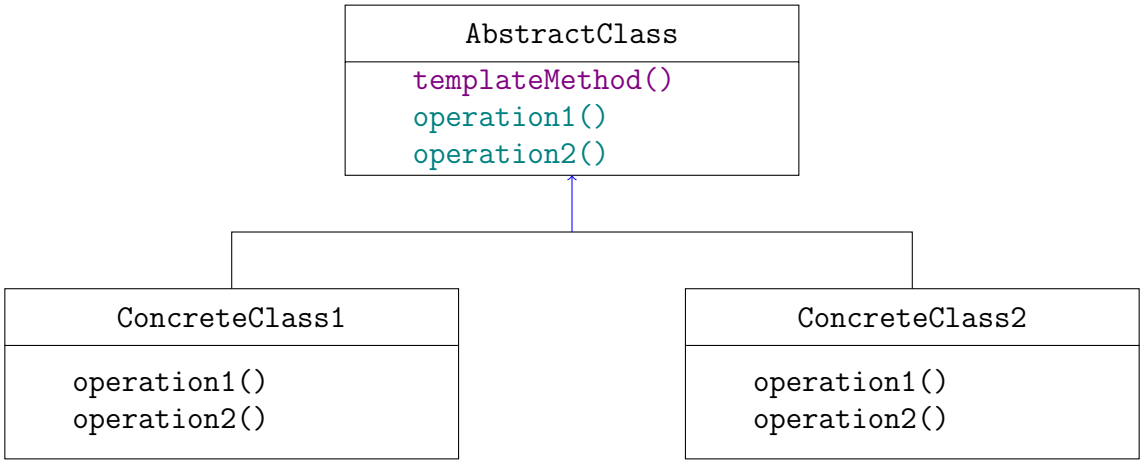
Now, everything that is common to the two algorithms is in the superclass, and only the part that is specialised is each individual class.

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.

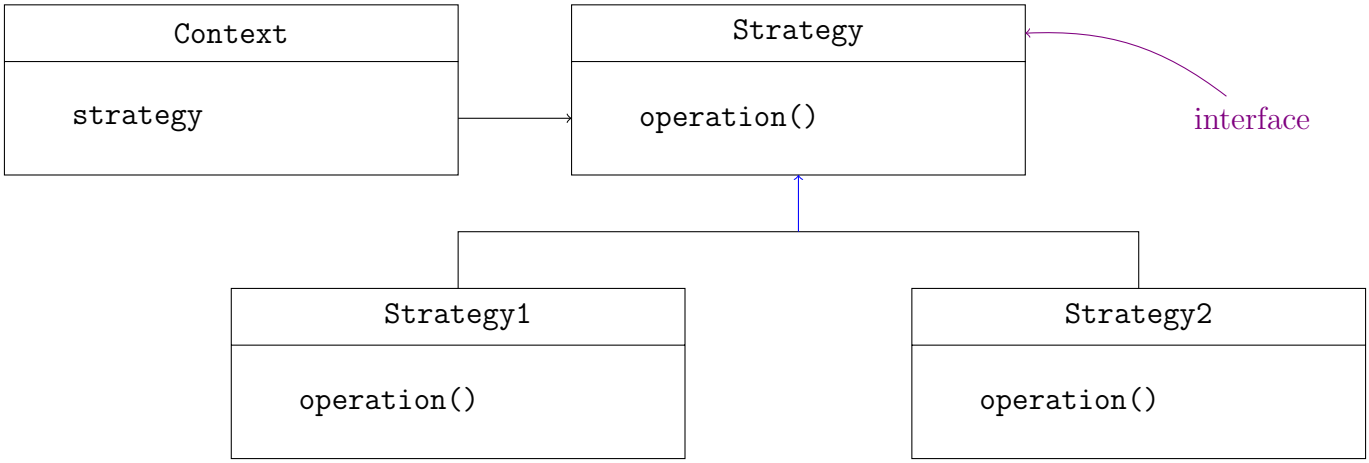
Coupling Metrics

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("...")
```

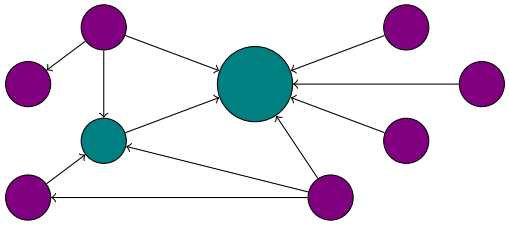
As the `Reverser` and `Doubler` no longer have a relationship with `Encoder`, we have must less coupling, hence we have better mobility and flexibility. Generally, composition should be preferred over inheritance.

25th October 2019

Stability

If a core object, such as a `Customer` class, has many things depending on it, such that it has a high afferent coupling, it can be a benefit since it can be heavily reused. However, it needs to be very stable, and require little to no changing, as changing it can break other parts of the codebase if done incorrectly. It should rely on very few other objects (if any). These are represented by the **teal** nodes below.

On other hand, the components at the edges (the ones in **violet**), can include UI components, such as a dashboard. It makes sense for the dashboard to depend on the `Customer` class, but not the other way around. These can be changed fairly easily, as not many things depend on it.



Complexity Metrics

Some methods for quantifying the complexity of a program are as follows;

- **cyclomatric (McCabe)**

  Described in McCabe's paper, it counts the possible different executions that could happen, and gives a lower bound on the number of tests required for a particular unit.

- **WILT (Whitespace Integrated over Lines of Text)**

  Integrates over the indented area of a given piece of code - this has a correlation with the McCabe complexity, but is much easier to calculate.

- **ABC (Assignments, Branches, and Conditions)**

  Every time it encounters each of the following constructs, it increments a counter by some number. This isn't as accurate as McCabe, but gives an approximation. For example, *Flog* is an ABC metric with specialist knowledge of Ruby;

```
1 def blah          # 11.2 total
2   a = eval "1+1" # 1.2 (a=) + 6.0 (eval) +
3
4   if a == 2      # 1.2 (if) + 1.2 (==) + 0.4 (fixnum) +
5     puts "yay"   # 1.2 (puts)
6   end
7 end
```

I honestly have no idea what this lecture is supposed to be going through. Generally, we should look at the trends for the complexity of code, rather than ensuring it is monotonically improving.

Turbulence

If we plot complexity against commits (number of changes), we can divide it into 4 general quadrants. Something that is high complexity, but has low commits, probably doesn't need to be refactored as we aren't modifying it often. This could be an algorithm copied from an academic paper, or a third party library being imported. The quadrant with low complexity and high commits could include configuration files, which are modified frequently.

The top right quadrant, which has high complexity and high commits should be a cause for concern. This is likely to introduce bugs as it is being changed often, and has high complexity.

29th October 2019

Factory Method Pattern

Factory methods (along with objects) allow for more flexibility and clarity with regards to object creation. Factory methods can have names unlike constructors. Note that the example on line 5 should be motivation to use factory methods, as it has less clarity, compared to the line above. By doing this, we can also close off some options by not exposing the constructor.

```
List<Object> l = Collections.emptyList();
Boolean b = Boolean.valueOf("true");
ExecutorService es = Executors.newFixedThreadPool(4);
VirtualMachine vm1 = VirtualMachine.withHardDiskGigabytes(128);
// VirtualMachine vm2 = new VirtualMachine(2, 512, 128);
```

The `VirtualMachine` example can be done as follows;

```
class VirtualMachine {

  private final int cores;
  private final int ram;
  private final int disk;
  private final int costPerMin;
  ...

  // factory methods are static
  public static VirtualMachine withHardDiskGigabytes(int size) {
    return new VirtualMachine(4, 128, size);
  }

  // private constructors force callers to use the factory methods
  private VirtualMachine(int cores, int ram, int disk) {
    this.cores = cores;
    this.ram = ram;
    this.disk = disk;
    this.costPerMin = (cores * ram + disk) / 50;
    ...
  }
}
```

Factory Objects

Sometimes, we don't know what we want to display until run-time (and therefore can't put it in directly) - as such, factory objects can decide what to return at run-time (example for some UI);

```
class LogoFactory {

  // can return any sub-type of Logo
  static Logo createLogo() {
    if (config.country().equals(Country.UK)) {
      return new FlagLogo("UK");
    }
    if (config.country().equals(Country.USA)) {
      return new FlagLogo("US");
    }
    return new DefaultLogo();
  }

  class FlagLogo implements Logo { ... }
  class DefaultLogo implements Logo { ... }
}
```

Therefore, when we need the logo, we can call `LogoFactory.createLogo()` to obtain the appropriate one. Additionally, if we want to add a new logo, it only needs to be modified in one place.

Abstract Factory Pattern

This is less frequently used. It's an interface of a factory, thus all implementations of it are factories.

```
interface WidgetFactory {
  Widget createScrollBar();
  Widget createMenu();
  ...
}

class MobileWidgetFactory implements WidgetFactory {

  @Override
  public Widget createScrollBar() {
    return new MobileScrollBar(Color.GREEN);
  }

  @Override
  public Widget createMenu() {
    return new MobileMenu(5);
  }
  ...
}

class DesktopWidgetFactory implements WidgetFactory {

  @Override
  public Widget createScrollBar() {
    ...
  }
  ...
}
```

This can help maintain consistency (such that we don't get a `MobileScrollBar` when building a desktop UI, etc).

Builder Pattern

Separate the construction of a complex object, likely with many parameters, from its representation. We also want the object to be in a "usable state" once it is first constructed, such that we don't have a transitionary state. An example is as follows;

```
public class BananaBuilder {

  private double ripeness = 0.0;
  private double curve = 0.5;

  // ensures the builder itself is created with factory method
  private BananaBuilder() { }

  public static BananaBuilder aBanana() {
    return new BananaBuilder();
  }

  public Banana build() {
    Banana banana = new Banana(curve);
```

```
15      banana.ripen(ripeness);
16      return banana;
17    }
18
19    public BananaBuilder withRipeness(double ripeness) {
20      this.ripeness = ripeness;
21      return this;
22    }
23
24    public BananaBuilder withCurve(double curve) {
25      this.curve = curve;
26      return this;
27    }
28  }
```

Note that the configuration methods return `this` (and is mutable), which gives a fluent interface allowing for method chaining. The builder can be used as follows;

```
1  import static BananaBuilder.*
2
3  public class FruitBasket {
4    ...
5
6    public FruitBasket() {
7      Banana banana = aBanana().withRipeness(2.0).withCurve(0.9).build();
8      ...
9    }
10 }
```

The `build()` method also allows us to validate the parameters (which can also have default values).

Singleton Pattern

We often do not need this pattern, and it can be troublesome when misused. The singleton pattern ensures a class only has one instance, and there is a global point of access - this is typically when instantiating the object has a high cost (high memory cost, actual fee, etc). An example of this is as follows;

```
1  public class BankAccountStore {
2
3    // static initialisation runs when class loaded
4    private static BankAccountStore instance = new BankAccountStore();
5
6    private Collection<BankAccount> accounts;
7
8    // private constructor ensures no-one can create a new instance
9    private BankAccountStore() {
10     ...
11   }
12
13   // getInstance() is called to gain access to the global instance
14   public static BankAccountStore getInstance() {
15     return instance;
16   }
17
18   public BankAccount lookupAccountById(int id) {
```

```
    ...
  }
  ...
}
```

The example above is an eager initialisation, which could be wasteful if it has a high cost to initialise, but no one actually needs it;

```
public class BankAccountStore {

  // no initialisation when class loaded
  private static BankAccountStore instance;

  private Collection<BankAccount> accounts;

  // private constructor ensures no-one can create a new instance (expensive)
  private BankAccountStore() {
      ...
  }

  // getInstance() is called to gain access to the global instance
  public static synchronized BankAccountStore getInstance() {
    if (instance == null) {
      instance = new BankAccountStore();
    }
    return instance;
  }

  public BankAccount lookupAccountById(int id) {
    ...
  }
  ...
}
```

Note that we have made the method thread-safe to avoid the race condition where multiple threads attempt to create instances simultaneously.

Dependencies

Note that in the example below, we have a strong coupling between a `PushSwitch` and a `Light`. This means that we cannot really test the switch independently of the light - which we want to avoid. Additionally, we cannot use this switch for another device, if required.

```
class Light:
  def on(self):
    print("Shining")

  def off(self):
    print("Dark")

class PushSwitch:
  def __init__(self):
    self.light = Light();
    self.on = False

  def press(self):
```

```
    self.on = not self.on
    if self.on:
      self.light.on()
    else
      self.light.off()

switch = PushSwitch()
```

The fix for this is following the **dependency-inversion principle** - we want to depend upwards on an abstraction, rather than downwards on a concrete class.

```
class PushSwitch:
  def __init__(self, device):
    self.device = device
    self.on = False

  def press(self):
    self.on = not self.on
    if self.on:
      self.device.on()
    else
      self.device.off()

  switch = PushSwitch(Light())
```

However, singleton creates dependency, and if we wanted to run unit tests, we would need the actual singleton instance in production (which may be sensitive, and cannot be tested on). It doesn't allow us to create mock objects.

1st November 2019

Legacy System

A legacy system is something that has been passed on, **and is of value**. While the examples covered assume good test coverage, this is rarely the case in actual systems, and we need techniques to apply our rules to existing codebases.

Additionally, code may be messy, or difficult to understand. As the codebase grows, it becomes infeasible to read every line of code to understand what it all does, and we need strategies to pick out what is useful, and what is dangerous to change. It's also important not to change code to "clean it up" when unnecessary, as it can be quite risky to completely rewrite working code.

We'd like to find out the connections between modules, how data flows in the system, and what changes could affect other modules. Sometimes, this can be done by having scripts visualise some diagrams, which displays how modules are connected.

Preserving Working Code

We want to avoid changing code that may look "broken" if we don't need to change it. A bug may be counteracted by another bug somewhere else in the system, and therefore "fixing" a bug in one place, may cause other parts of the system to break. Furthermore, this "incorrect" behaviour may be used by the customer, or the end user, and they may rely on it working in the same way.

Confidence and Testing

In some code bases, it may be difficult to make changes, or it may be higher-risk in an environment which deals with finances.

To build confidence on modifying code, we want to have some form of guidance that our changes haven't broken any functionality. We can split tests into two types;

- **unit test** micro level

  This can give a lot of information about a small bit of code - it's not feasible to write this for all of a legacy system.

- **system test** macro level

  At a system level test, we run an expected action all the way through, and track what is called. If something fails, it doesn't tell us where, just that the functionality is incorrect.

Generally, we start by adding unit tests for the small region of code we are changing.

Examples of code that may be difficult to test are generally environment dependent - if a piece of code in production relies on making requests to a (protected) server, or a database, accessing this may be difficult or give inconsistent results, whereas we'd ideally want to test in isolation.

Seams and Sensing

The idea of a **seam** is a place where behaviour can be altered without editing in that particular place. For example, if we wanted to test something that has a web endpoint, we wouldn't want to change it, test it, change it back, and then commit it (since we shouldn't change code after testing). A seam allows us to inject different behaviour. Every seam has an enabling point, were we can make the decision to use one behaviour or another.

We break dependencies to **sense** when we can't access values our code computes.

Live Example

See Panopto timestamp 34:50 for this. Notes will be added here, in bullet points;

- we don't want to change too much, as we can't test if it works in the same way as before, but we do have to change a bit to test it
- we want to simulate the behaviour when a website is broken, but this generally isn't the case
- we extract the behaviour of `WebPageProbe` to implement the interface `Probe`, which we can make always fail
- we can then create a **new** constructor (don't modify existing constructor), which takes in the failing `Probe`
- this has the expected behaviour, but we are now notifying on every test
- we then spot a use of `getInstance()`, suggesting a singleton pattern; we can try taking this out into a field, and adding it to the constructor
- we now have the **enabling point** in the test suite, allowing us to swap out the `Alerter` (interface for `Emailer`)
- this behaviour can either be tested by manually implementing fields in our implementation of `Alerter`, or with a mock object library
- we have a dependency on the system clock, which we can try to control in a similar way to allow for testing time based behaviour

5th November 2019

Threads

Note that when we create new threads, they should implement `Runnable`, and override `run()` (since we aren't really specialising the thread's behaviour);

```
public class ThreadExample {
  public static void main(String[] args) {
    new Thread(new Task("A")).start();
    new Thread(new Task("B")).start();
  }
}

class Task implements Runnable {

  private final String name;

  public Task(String name) {
    this.name = name;
  }

  @Override
  public void run() {
    print("starting thread " + name);
    ... // some code with possible delays
    print("completed thread " + name);
  }
}
```

Note that extending `Thread` is a use of the template pattern, whereas implementing `Runnable` is a use of the strategy pattern, thus avoiding the coupling. By doing this, we can also run `Task` synchronously, with `run()`, without needing it to be in a `Thread`.

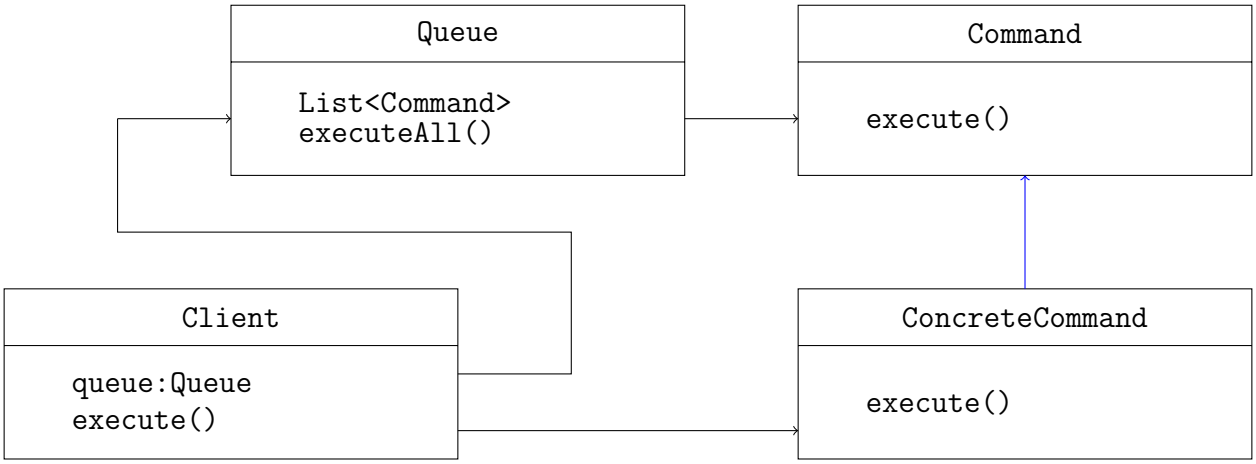
Callables

Note that a `Runnable` has no return value, whereas a `Callable` does. This can be used if we want to perform some computation asynchronously;

```
class CountingTask implements Callable<Integer> {

  private final String name;

  public CountingTask(String name) {
    this.name = name;
  }

  @Override
  public Integer call() {

    print("starting callable " + name);
    ... // some code with possible delays
    print("completed thread " + name);
    return 42;
  }
}
```

Command Pattern

This pattern wraps up a piece of behaviour in an object, which we can be done now, later, or repeatedly. These operations can be queued or undone. A command is an instruction to do something. It can be thought of passing around a function (which is the operation), and a state (containing options).



For example, we can have the following;

```
1  public class QueuingExample {
2    public static void main(String[] args) {
3
4      CommandQueue queue = new CommandQueue();
5      queue.add(new Command("A"));
6      queue.add(new Command("B"));
7
8      queue.runAllCommands();
9    }
10 }
11
12 class CommandQueue {
13
14   private Queue<Runnable> queue = new LinkedList<Runnable>();
15
16   public void add(Command command) {
17     queue.add(command);
18   }
19
20   public void runAllCommands() {
21     for (Runnable command : queue) {
22       command.run();
23     }
24   }
25 }
26
27 class Command implements Runnable {
28
29   private final String name;
30
31   public Command(String name) {
32     this.name = name;
33   }
34
```

```
35    @Override
36    public void run() {
37      System.out.println("Starting command " + name);
38      ...
39      System.out.println("Finishing command " + name);
40    }
41  }
```

However, all of this is currently running in a single thread. We can think of an `Executor` as a command queue which has a set of threads (a pool), which can each perform a task.

We can think of a function that adds tasks to the queue as **producers**, and the threads as **consumers**, who "compete" for tasks. Queues are also load balancers, as it can deal with different lengths of jobs naturally, as a job is picked up by a thread as soon as one is free. Therefore producers can enqueue in bursts, and consumers can work steadily.

```
1  public class CommandExample {
2
3    public void run() {
4      // gives us a pool of 3 threads
5      Executor queue = Executors.newFixedThreadPool(3);
6
7      for (int i = 1; i <= 10; i++) {
8        // note that Command implements Runnable
9        queue.execute(new Command(i));
10     }
11   }
12 }
```

Futures

However, we might want to to obtain a calculated response from our tasks. By using `Future`s, we can get the result once it is finished.

```
1  public static void main(String[] args) throws Exception {
2
3    MyCallable task = new MyCallable("A");
4
5    ExecutorService executor = Executors.newFixedThreadPool(2);
6
7    Future<Double> future = executor.submit(task);
8
9    // do something whilst calculating
10
11   // the following line blocks until the result is available
12   Double result = future.get();
13 }
```

By using this `Future`, that's returned after submitting the task to the `ExecutorService`, we can do the following;

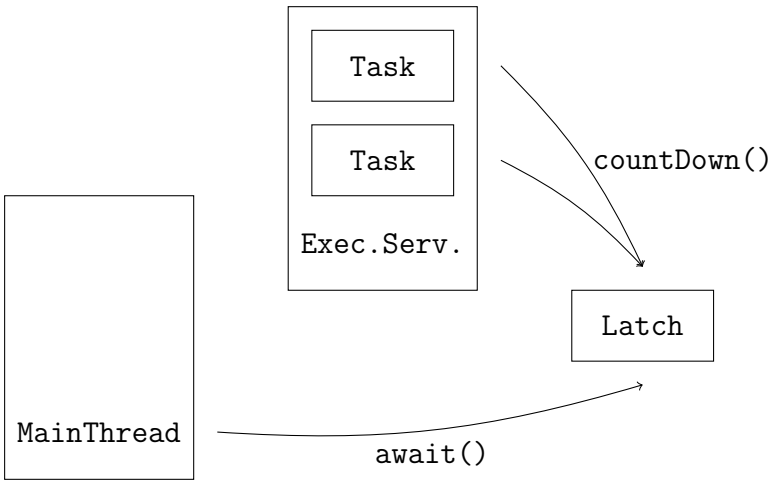
- `get()` blocks until ready
- `get(timeout)` throws exception if it doesn't complete in time
- `cancel(mayInteruptIfRunning)` the parameter allows a running task to be cancelled
- `isDone()` is it done

- `isCancelled` is it cancelled

```
public static void main(String[] args) {

  ExecutorService executorService = Executors.newFixedThreadPool(2);

  executorService.submit(new MyTask("A"));
  executorService.submit(new MyTask("B"));
  executorService.submit(new MyTask("C"));
  executorService.submit(new MyTask("D"));

  // do not accept more tasks
  executorService.shutdown();

  try {
    // wait a maximum of 2 minutes for everything to terminate
    executorService.awaitTermination(120, TimeUnit.SECONDS);
  } catch (InterruptedException e) {
    // handle the case where it times out
  }
}
```

Latch

We don't want to shut down the `ExecutorService` once all the jobs are completed, but we want to cluster jobs together, such that the main thread can continue once a part of it is done. For this, a `Latch` can be used, which is set with an initial value, and then decremented by the tasks.



A simple example is as follows;

```
public class LatchExample {
  public static void main(String[] args) throws Exception {

    ExecutorService executor = Executors.newFixedThreadPool(2);
    CountDownLatch latch = new CountDownLatch(4);

    executor.submit(new LatchedTask("A", latch));
    executor.submit(new LatchedTask("B", latch));
    executor.submit(new LatchedTask("C", latch));
    executor.submit(new LatchedTask("D", latch));

```

```
12      latch.await();
13    }
14  }
15
16  class LatchedTask implements Runnable {
17
18    private final String name;
19    private final CountDownLatch latch;
20
21    public LatchedTask(String name, CountDownLatch latch) {
22      this.name = name;
23      this.latch = latch;
24    }
25
26    @Override
27    public void run() {
28      System.out.println("Starting " + name);
29      // do something
30      System.out.println("Finished " + name);
31
32      latch.countDown();
33    }
34  }
```

Note that the `latch` is shared by each batch of tasks. At the end of running, each task must `countDown()`. Generally, we should use executors instead of creating our own threads.

12th November 2019

Graphical User Interfaces

Typically, for whatever platform we're working on, there should be a library with components that we are able to use for our interface. This reduces the work we have to do significantly, as we don't have to write all the UI components from scratch.

Example

This lecture uses *Swing*, which is a fairly simple library, and is quite outdated.

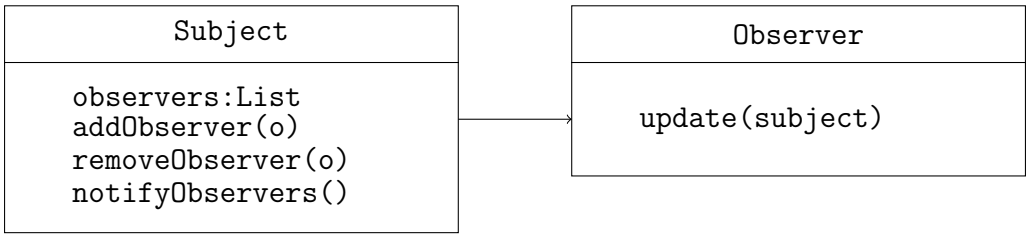
```
1  package interactive;
2
3  import ...;
4
5  public GuiApp {
6
7    public class GuiApp {
8      new GuiApp.display();
9    }
10
11   private void display() {
12     JFrame window = new JFrame("Example App");
13     window.setSize(400, 300);
14
15     JPanel panel = new JPanel();
16
```

```
17      // 10 character limit
18      JTextField textField = new JTextField(10);
19      JButton button = new JButton();
20
21      // this can be replaced with a lambda
22      button.addActionListener(new ActionListener() {
23        @Override
24        public void actionPerformed(ActionEvent e) {
25          textField.setText("clicked");
26        }
27      });
28
29      panel.add(textField);
30      panel.add(button);
31
32      window.add(panel);
33
34      window.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
35
36      // do this at the end, once everything is set up
37      window.setVisible(true);
38    }
39  }
```

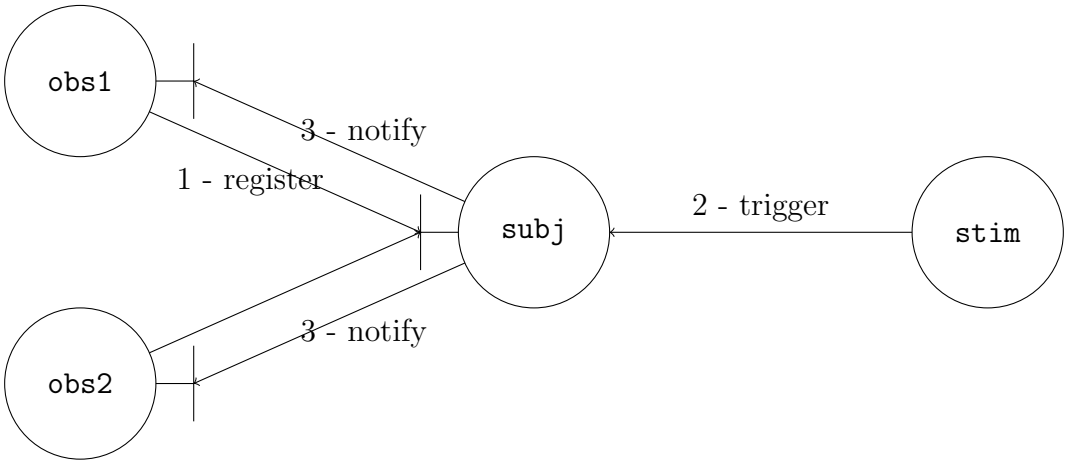
Observer Pattern

In a non-interactive application, the flow is fairly simple - it read from the top down, branching or looping if needed. However, in an interactive application, we need to react to **events** from the user. When something happens, something else may happen (triggered by the first event).

The observer pattern has objects that know when another object's state is changed - by being told, not by continuously querying whether the event is complete. This is also known as registering a callback, or publish-subscribe.



When the state of the `Subject` has changed, all of its `Observer`s can be notified via `update` with its state. Note that an observer must first register to be notified;



Model-View-Controller (MVC)

MVC splits interactive apps into 3 parts - data, display, and user input. The **view** is concerned with drawing on the screen (anything presentational or to do with formatting). Front-end specialists can work on this part. This separation allows for different views, without changing the model.

Model is to do with the logic of the application, and modelling the domain. Finally, the **controller** is concerned with events - it should be kept simple, as it is a translation from the world of events. Typically, all the logic should be in the model, and not in the controller.

An example of code that doesn't follow this **architectural pattern** is as follows;

```
public class GuiApp {

  private final JButton button = new JButton("Press me!");
  private final JTextField textField = new JTextField(10);

  private final PressCounter pressCounter = new PressCounter();

  class PressCounter {

    private int count;

    public void increment() {
      count++;
      if (count < 5) {
        textField.setText(String.valueOf(count));
      } else {
        textField.setText("Too many!");
        button.setEnabled(false);
      }
    }
  }

  private void display() {
    ... // standard setup

    button.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent actionEvent) {
        pressCounter.increment();
      }
    });
    ...
  }
}
```

We can separate this into the following classes. Note that we are using the **observer pattern**, as we then query the `Model` passed in.

```
class View implements Updatable {

  private final JButton button = new JButton("Press me!");
  private final JTextField textField = new JTextField(10);

  public View(ActionListener controller) {
    JFrame frame = new JFrame("Example App");
    frame.setSize(400, 300);
```

```
9
10      button.addActionListener(controller);
11
12      JPanel panel = new JPanel();
13      panel.add(textField);
14      panel.add(button);
15      frame.add(panel);
16      frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
17      frame.setVisible(true);
18    }
19
20    public void update(Model model) {
21      if (model.morePressesAllowed()) {
22        textField.setText(String.valueOf(model.count()));
23      } else {
24        textField.setText("Too many!");
25        button.setEnabled(false);
26      }
27    }
28  }
```

This contains all the logic, and also contains functions that can easily be called from the `View`. Since this isn't specific to the display, it can be reused in other applications easily. We can also add more observers (`View`s) to the model - by having any number of observers, we can also have none, which allows for unit tests to be easily written.

```
1  class Model {
2
3    private final Updatable view;
4
5    private int count;
6    private boolean morePressesAllowed = true;
7
8    public Model(Updatable view) {
9      this.view = view;
10   }
11
12   public void increment() {
13     count++;
14     if (count >= 5) {
15       morePressesAllowed = false;
16     }
17     view.update(this);
18   }
19
20   public boolean morePressesAllowed() {
21     return morePressesAllowed;
22   }
23
24   public int count() {
25     return count;
26   }
27 }
```

Finally, the controller (and wiring) is as follows;

```
1  public class GuiApp {
2
3    private View view = new View(new Controller());
4    private Model pressCounter = new Model(view);
5
6    class Controller implements ActionListener {
7      public void actionPerformed(ActionEvent actionEvent) {
8        pressCounter.increment();
9      }
10   }
11
12   public static void main(String[] args) {
13     new GuiApp();
14   }
15 }
```