1) TDD Process:

1) Make tests checking for all the behaviours we want. 2) Add the minimum amount of code required to make us pass these tests. If our code doesn't do what we want then it is a sign that we don't have enough tests and behaviours.

2) Coupling:

Two bits of code are coupled when they must change together. High coupling causes classes to be **immobile** and we can't use the classes independently. Two types of coupling:
1) **Afferent Coupling**: How many other classes use this class - a measure of this class's responsibility
2) **Efferent Coupling**: A measure of how many different classes are used by this class - a measure of this class's independence. Modifying Afferent Coupled classes is dangerous, but not dangerous for Efferent ones.
Inheritance gives us a strong coupling between sub and superclasses. If a class knows a lot about the inner implementation of another and uses this, then it is more strongly coupled.

3) Mock Objects:

Mock objects are simulated objects that mimic the behaviour of real objects in controlled ways. We use these to create a **false instance of a class we own**. The main purpose of mock objects is to allow developers to test how a system interacts with its dependencies without having to rely on the **actual implementations of those dependencies**. We mock **interfaces**.

```
import
org.jmock.integration.
junit.JUnitRuleMockery;
import org.junit.Rule;
import org.junit.Test;
public class HeadChefTest {
@Rule
public JUnitRuleMockery mock =
      new JUnitRuleMockery();
static final order APPLE_TART =
      new Order("apple_tart")
static final order SOUP =
      new Order("soup")
Chef pastryChef =
      context.mock(Chef.class)
@Test
public void delegatesDessertsTo
PastryChef() {
HeadChef headChef = new
HeadChef();
context.checking(new
Expectations() {{
exactly(1).of(pastryChef).order(
APPLE_TART);
}
headchef.order(SOUP, APPLE_TART)
```

The pastryChef is a collaborator who should be receiving orders from the Head Chef. We are testing purely the behaviour of the head chef – if they are sending the correct messages to the pastry chef. Therefore, we use the mockery (context) to create a mock Chef to act as the Pastry Chef, and use the real implementation of the Head Chef, to see if the Head Chef (that we want to test) is sending orders properly to our pastryChef. `ignoring()` is a very useful method to have in our expectations block – it means that we still pass the test even if we invoked methods that we did not expect to. We can force our mock objects to return a specific value with `will(returnValue(3))` after an "exactly statement".

4) The Template Method Pattern:

We want to develop our code now to have less duplication. The idea: this pattern defines the steps of an algorithm **as separate methods, allowing subclasses to provide their own implementations for some or all of the steps.** The main advantage of this pattern is that it allows you to define the **overall structure of an algorithm, while allowing subclasses to customize or override specific steps as needed.** Here is an example in Java:

```
abstract class foo {
  public void templateMethod() {
    step1();
    step2();
    step3();}
  protected void step1() {}
  protected void step2() {}
  protected void step3() {}}
class bar extends foo {
  @Override
  protected void step2() {…
   /*some custom code for step 2*/
  }
}
```

We have an abstract superclass. It contains all of the generic code (that is carried out in all of the variations of the method). However, those methods which are different (for a specific implementation (subclass) of the abstract class) are overridden, and an implementation for them is held in the subclass.

5) The Law of Demeter

Bad design = rigid, immobile, fragile._The Law of Demeter is a design principle that specifies the ways in which an object or module should communicate with other objects. "**An object should only communicate with its immediate neighbours**." This means that an object should only send messages to / request services from the objects that it directly communicates with, rather than calling methods on objects that it has obtained through other means (e.g, if it has some field A which has some field B, calling A.B.method() is not good). Following this helps to reduce the number of dependencies between objects, which makes the system easier to **understand and maintain** because it decouples the objects from one another and makes them easier to modify or replace (changing classes could break many others if we violate the Law of Demeter a lot).

6) The Strategy Pattern

The strategy pattern is a behavioural design pattern **that enables an algorithm's behaviour to be selected at runtime**. It involves creating a family of algorithms**, encapsulating each one, and making them interchangeable**. The client can then choose which algorithm to use based on the context. This is how we do it in Java: 1) We make an interface that is implemented by all of the subclasses, and **have the methods that differ declared within this interface**. 2) Then, we make a superclass that **contains all of the commonalities of those subclasses** 3) We instantiate this superclass **with an argument in the constructor being the subclass** (have a field of the interface type). Then, for the methods that need to differ in the superclass, we just **call the interface field.method() in it**.

```
interface Strategy {int execute(int a,
int b);}
class AddStrat implements Strategy {
  @Override
  public int execute(int a, int b)
{return a+b;}}
class SubStrat implements Strategy {
  @Override
  public int execute(int a, int b)
{return a-b;}}
class Context {
  private Strategy strat;
  public Context(Strategy s) {this.strat
= s;}
  public int executeStrategy(int a, int
b) {return strategy.execute(a, b);}}
```

7) The Factory Pattern:

In the factory method pattern, we have a factory object that creates objects that are some subclass of the factory method's output type **based on some input data.**

```
public interface Animal extends
Producible {void makeSound();}
public class Dog implements Animal {
  public void makeSound(){
    System.out.println("Woof!");}}
public class Cat implements Animal {
  public void makeSound() {
    System.out.println("Meow.");}}
public class AnimalFactory implements
Factory {
public Animal produce(String animal) {
   if (animal.equals("Dog")) return new
Dog();
   else if (animal.equals("Cat")) return
new Cat();
   else throw new
IllegalArgumentException();}}
public interface Producible{}
public interface Factory{
  public Producible produce(String
specifier);}
```

This is useful when: 1) We want to instantiate objects based on data at runtime 2) We want to cache objects (might be expensive to instantiate some things) 3) Lets us isolate object creation logic. We could also have some Factory interface, that factories implement, as seen above.

8) The Builder Pattern:

The builder pattern is a design pattern that allows for the creation of complex objects using a **step-by-step approach**. It is particularly useful when the construction of an object involves many steps or requires complex logic, and you want to hide this complexity from the client code that uses the object. In the builder pattern, **a builder class is used to represent the construction process of an object**.
The builder class has methods for configuring the various parts of the object, and a separate build method that creates the object based on the current configuration. The client code uses the builder class to create the object step by step, calling the various configuration methods as needed.

```
public class Computer {
   private String processor; private
String memory; private String storage;
   private Computer(ComputerBuilder
builder) {this.processor = builder.p;
        this.memory = builder.m;
        this.storage = builder.s;}
   public static class ComputerBuilder {
     private String p; private String m;
     private String s;
     public ComputerBuilder withP(String
p){this.p = p; return this;}
     public ComputerBuilder withM(String
m){this.m = m; return this;}
     public ComputerBuilder withS(String
s){this.s = s; return this;}
     public Computer build()
  {return new Computer(this);}}
 public static void main(String[] args) {
   Computer c = new Computer.
ComputerBuilder().withP("i9").withM("2TB"
).withS("12PB").build();}}
```

9) The Singleton Pattern:

The singleton pattern is a design pattern that ensures that a class has only one instance, and provides a global access point to that instance. In the singleton pattern, a class has a private constructor and a static method that returns an instance of the class. **This static method should be synchronised to avoid race conditions on multiple versions of the singleton being instantiated**. The first time the static method is called, it creates a new instance of the class and returns it. Subsequent calls to the static method return the same instance. This ensures that there is only ever one instance of the class, and provides a global point of access to that instance. Allows for lazy creation. However, it becomes very difficult to test the class if there is no seam where we choose if we can use the singleton or not.

10) Dependency Inversion

We make a class no longer rely on a concrete class but on an abstraction instead. For example, this class is too dependent on the Light class.

```
class PushSwitch {
   Light light = new Light();
   boolean on = false;
   public PushSwitch() {}
   public void press() {
       if (on) light.off();
       if (!on) light.on();
       on = !on;}}
```

In Java, we fix it by simply making an interface that implements the methods we need of our concrete class, and passing a reference to the concrete class we need. We call the methods on this object passed in. To detect violation of this pattern is to see if we specifically mention the name of an object/class in our use of the program.

```
class PushSwitch {
 Switchable device; boolean on =
false;
 public PushSwitch(Switchable
device) {
   this.device = device;}
 public void press() {
   if (on) device.off();
   if (!on) device.on();
   on = !on;}}
interface Switchable {
   public void on();
   public void off();}
```

This can also be used to make testing easier (passing in a Seam so that we don't execute on the real System – an issue the Singleton pattern might encounter), as it gives us flexibility to use many different implementations.

11) The Command Pattern

The classical way of doing Concurrency in Java is by making classes extend Thread and have a .start() method, or implement Runnable and have the .run() method. **The Command Pattern** involves a command queue – where we queue up our commands, and run .executeAll() to execute them. Having some producer threads queue up commands, and some consumer threads process them improves performance as it acts as a load balancer.

```
import
java.util.concurrent.Executor;
import
java.util.concurrent.Executors;
public class Execution {
public static void main(String[]
args){
 Executor executor = Executors.
 newFixedThreadPool(2);
 executor.execute(new C("A",10));
 executor.execute(new C("B",10));
 executor.execute(new C("C",10));
 System.out.println("Done.");}
public static class C implements
Runnable {
  String name; int cTo = 0;
  public C(String n, int c){
  this.name = n; this.cTo = c;}
  @Override
  public void run() {
    System.out.println("name: " +
     name + " count: " + cTo);
    cTo++;}}}
```

An **ExecutorService** is an **Executor** but with more methods. We could have also **implemented java.util.concurrent .Callable** instead of Runnable, which allows us to return a value with .call(). To get our values we use a **Future** (Futures have a few methods like .get()):

```
ExecutorService e = Executors.
newFixedThreadPool(2);
Future<Integer> f =
executor.submit(new Cnt("A",10));
```

Cnt is a class which implements Callable

12) Futures and Latches

A **Future** represents the result of an asynchronous computation. It acts as a **placeholder for the result of a task that has not yet been completed**. You can use a Future to **check the status of a task, cancel a task, or retrieve the result of a completed task**. You can also use a Future to block the current thread until the task has completed, if desired. **Methods**: `get()`, `get(timeout)`, `cancel(boolean mayInterruptIfRunning)`, `isDone()`, `cancelled()`. A **latch** is a concurrent utility that allows one or more threads to **wait for a set of operations to complete**. A latch is **initialised with a count, and the count is decremented each time the latch is released - when a thread finishes its task**. When the count reaches zero, the latch is considered to be "open", and any threads that are waiting for the latch to open will be released. For Java, we want to decrement our latch in our run/call method. We initialise it at the number of processes to wait for.

```
public static void main(String[] args)
throws InterruptedException {
 int threadsToWaitOn = 4;
 CountDownLatch latch = new
  CountDownLatch(threadsToWaitOn);
 ExecutorService executorService =
Executors.newFixedThreadPool(threadsToWai
tOn);
 /* ... do the execution ... */
 latch.await();}
```

13) Interactive Applications

This is what a simple swing app looks like:

```
import javax.swing.*; public class
swingApp {
 public static void main(String[] args) {
   new swingApp().display();}
 private void display() {
     JFrame frame = new JFrame("Example
App"); frame.setSize(400, 300);JPanel
panel = new JPanel(); JTextField tf = new
JTextField(10);JButton b = new
JButton("Press"); panel.add(tf);
panel.add(b);frame.add(panel);
frame.setVisible(true);
frame.setDefaultCloseOperation(WindowCons
tants.EXIT_ON_CLOSE);}}
```

13.1) The Observer Pattern

The **Observer Pattern** is a software design pattern in which an **object**, **called the subject** (which we want to observe), maintains a list of its dependents, **called observers**. When a change of state happens, it notifies these observers which act accordingly. It is mainly used to implement distributed event handling systems, e.g. GUIs. For GUI objects we do this by adding ActionListeners in the Display:

```
b.addActionListener(e->tf.setText("K"));
```

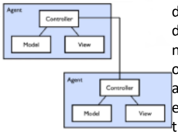
We may want multiple Observers on our Display, in which case we can have an addObserver method which adds to our list of observers. Any changes then have all observers notified.

13.2) The MVC Design Pattern

This is a software design pattern in GUIs that **separates the representation of information from the user's interaction with it.** Model = application data and business logic, View = presentation of the data, Controller = input handler, converting input into commands for the model or view. It updates the model and view components as needed. To implement it in Java, we'd typically have 3 different classes representing each of the three. We'd wire up these classes as follows:
1) The View contains the code for displaying the UI and nothing else. For each of the things required to sense input, we initialise them with **actionListeners()** here, pointing to a method in the controller. The **view contains a reference to the controller.** 2) The Controller detects input and sends it to the correct method in the Model to be processed. Thus, the **controller has a reference to the Model.** 3) The model computes everything and stores any data. Any updates are sent back to the View (**it is an observer of the Model**)

13.3) PAC Pattern

This design pattern is a software architecture pattern that separates the **presentation layer (user interface)** from the **abstraction layer (business logic & data manipulation)** and **the control layer (workflow of application & flow of information).** This allows the separation of concerns between these different layers, making it easier to design and maintain the software. It allows developers to more easily update and maintain the user interface without affecting the underlying business logic or workflow of the application. The way this is setup is we have many small MVC setups, each called an **agent**. Each **agent** only looks after a particular part of the UI.



Since each agent controls a different part of the UI and different parts of the UI may need to interact with each other, the agents must be able to communicate with each other. They do this through their controllers. This models the hierarchical structure we see in UIs. An agent can only communicate one level up or down, so if we need to communicate between two "distant" agents, this takes many resources. We can fix this by using an **Event Bus**. Instead we have all our agents but they don't form a tree, instead we send publish our Event to an Event Bus, and all agents are subscribed to the Event Bus. The relevant Agent can be notified then as a result if a message needs to be communicated to it. This is more efficient, but **we lose the hierarchical nature our tree gave us.**

14) System Integration

When we want to combine code from different systems with that of our own, or code of our peers, we carry out System Integration. A number of patterns can help us.

14.1) The Adaptor Pattern

The Adapter pattern lets us use otherwise **incompatible classes** by providing interfaces compatible with the client. We have a class called Square. We want to use this class **in a program that expects a Rectangle**, but the Rectangle **interface does not have a method for setting the length of all sides at once like the Square class does**. Adapters fix this:

```
public interface Rectangle {
   public int getWidth();
   public int getHeight();
   public void setWidth(int height);
   public void setHeight(int
height);}
public class SquareToRectangleAdapter
implements Rectangle {
   private Square square;
   public
SquareToRectangleAdapter(Square
square) {this.square = square;}
   @Override public int getWidth()
{return square.getSideLength();}
   @Override public int getHeight()
{return square.getSideLength();}
   @Override public void setWidth(int
width) {setHeight(width);}
   @Override public void
setHeight(int height)
{square.setSideLength(height);}
 public class Square {
       int sideLength;
       public Square(int sideLength)
this.sideLength = sideLength;}
       public int getSideLength()
{return sideLength;}
       public int setSideLength(int
length) {this.sideLength = length;}}}
```

This allows the program to use the SquareToRectangle Adapter class as if it were a Rectangle object, while having the functionality provided by the Square class. We use adapters when we want to insulate our implementation from a third party library (they can change unexpectedly). Makes our code more flexible - we could switch over to a different implementation more easily.

14.2) The Decorator Pattern

The Decorator pattern is a design pattern that allows you to add **new behaviour to existing objects dynamically.** It is an alternative to subclassing, which involves creating a new class that is a modified version of an existing class. We create a wrapper class that "decorates" the original class by adding new behaviour. The wrapper class contains a reference to the original class and delegates method calls to it, while also adding new behaviour before or after the method call.

14.2) The Facade Pattern

The facade pattern is a design pattern that provides a simplified interface to a complex system of classes, libraries, or frameworks. We hide away complexities to make the interface easier to use. The facade pattern is useful when you want to **provide a simple interface to a complex system**, or when you want to **decouple a client from the implementation details of a subsystem**. It can also be helpful for reducing the number of dependencies between classes, which can make it easier to maintain and test your code. In the facade pattern, the facade class acts as a wrapper for the subsystem classes. It provides a simpler, easier-to-use interface to the subsystem, while still allowing the client to access the subsystem classes directly if needed. The facade class may also provide additional functionality that is not available in the subsystem classes.

14.3) The Simplicator Pattern

Similar to the Façade Pattern. We put an interface in front of a more complicated one, and standardise / simplify what is contained in the old interface to make it more easy to use.

14.4) The Proxy Pattern

The proxy pattern is a design pattern that provides a surrogate or placeholder object that controls access to another object. **The proxy object controls access to the original object**, and can be used to add additional functionality or behaviour when accessing the original object. The original object might be expensive to access, require security clearance, we might want to load balance or do caching. We share an interface between the real system and the proxy. The proxy may or may not delegate to the real service on a request.

14.5) Hexagonal Architecture / Ports and Adaptors

This pattern promotes the separation of concerns between the business logic of a software application and its external dependencies. In hexagonal architecture, the business logic of the application is contained within the core of the application, and is surrounded **by a layer of adapters** that allow the application to **communicate with external dependencies** such as databases, user interfaces, and external APIs. This is based on the idea that the business logic of an application should be independent of the specific details of the external dependencies it interacts with. This makes it easier to test and maintain the application, and allows the application to be more flexible and adaptable to change. The code which is specific to the third party library is only on the outer layer, not polluting the business logic. If we need to work with a different third party library then **we can just use a different adapter**.

14.5.1) Testing our Hexagonally Structured System

We can test the entire core with **Unit Tests**. We test the adapters layer and communication with outside systems with **Integration Tests** (which test interaction between different components/modules of system). We can test our entire application with **System** Tests which invoke the full system and interact with the third party, ie: we can perform a real transaction end to end across the system. Takes a lot of time and we don't get good specialised feedback as to what failed – just that something did fail, unlike Unit Tests. Integration Tests are essential to make sure everything works end to end. Ideal test coverage is 70% Unit Tests, 20% Integration Tests, 10% System Tests.

15.1) Services

Services - software components that can be downloaded and assembled into new systems. Splitting services into multiple components, helps organisation and parallelizing computation. Of our components some provide services, others are clients of them. Microservice architectures where apps are built from many small services that together form the full system rather than a monolithic app are ideal. We can use graphs to visualise how all the services interact. The web itself has evolved into a distributed system with many microservices.

15.2) REST

REST services are built around the idea of resources and representations. Resources are things in the world, physical or conceptual, identified by URIs - Uniform Resource Indicators (similar to URLs – in the URL https:// football/player/robin, Robin indicates a specific player). Transferring resource representations allows us to communicate between services. We can have multiple representations of the same resource e.g. XML / JSON. Binary formats exist too which are platform independent, though we'd need a parser at both ends to send data between different systems in different languages. The **Richardson Maturity Model** categorises webservices based on how much they use URI for classifying resources, HTTP methods, and Hypermedia. **1) Level 0**: Uses HTTP to transport data, doesn't use URIs to identity resources, different HTTP methods to describe actions, or hypermedia. A single URI is used to identify the service "endpoint", to which requests are posted. In a Level 0 service, each request consists of a document or a set of parameters that describes the request. All requests generally go to the same URI by the same method. The service parses the request document to determine what the client wants. An example of this is using a SOAP (Simple Object Access Protocol) envelope to wrap an XML document describing a request. Technologies such as WSDL attempt to describe the protocol expected by a service, and describe the format of request/response documents. **2) Level 1** services make use of more URIs to represent different types of resources in the system, but typically do not take advantage of all of the available HTTP methods. They also do not respect the correct semantics for HTTP methods like GET, as they often use GET requests to cause side-effects on the state of the system. **3) Level 2** services use URIs to represent different types of resources, and also respond to different HTTP methods (GET, POST, PUT and DELETE) to update the state of these resources. They send appropriate HTTP status codes with their responses, which allow the client to track the effects of the calls they have made. **4) Level 3** is the highest level in the Richardson model, and characterises fully RESTful services. It builds on Level 2, with the same ideas of identifying resources using URIs, and acting upon these resources using different HTTP methods. The key point about Level 3 services is that the representations that they use contain hyperlinks to other resources that the client can follow. With Level 2 services, clients often follow URI templates to construct the URI for a particular resource. In Level 3 services we might do a search, and then get back a document containing links to the user records, which we can follow, without having to assume the structure of those links.

17) Map Reduce:

Map Reduce is a way of distributing computation to speed the entire process up. In the example of summing squares [1, 2, 3, 4] we can do this by mapping some (get [1,4], [9, 16]), reducing some (get 5, 25), then reducing these to get our final answer (30). This can clearly be parallelized. This process inspired Hadoop. Mapper:

```
public class Map extends Mapper {
  private final static IntWritable one =
new IntWritable(1);
  private Text word = new Text();
  public void map(LongWritable key, Text
value, Context context) throws
IOException, InterruptedException {
      String line = value.toString();
      StringTokenizer tokenizer = new
StringTokenizer(line);
      while (tokenizer.hasMoreTokens()) {
        word.set(tokenizer.nextToken());
        context.write(word, one);
     }}}
```

The type parameters on Mapper reflect the types of k1, v1, k2 and v2. context. We emit a pair from the mapper by calling context.write(word, one). Num output pairs from the mapper isn't necessarily equal to num input pairs. Reducer:

```
public class Reduce extends Reducer {
  public void reduce(Text key, Iterable
values, Context context) throws
IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      context.write(key, new
IntWritable(sum)); }}
```

Depending on the complexity of each phase, we could have more Reducers than Mappers. Reducers can be chained if the reduce function is commutative and associative. This can lead to improved efficiency as the initial reductions can be done nearer to the mappers, so we don't have to move so much data around. The magic of MapReduce is in the shuffle. This is where all of the key-value pairs output from the map for a given key value are gathered up and supplied to the same reducer, so that it can do its work. An example use of MapReduce could be a distributed grep - I want to grep for a certain word in a very large file: gigabytes or terabytes of data. The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

18) Working with Legacy Systems:

The first thing we should do is discover the dependencies of objects. We can visualise the dependencies using diagrams. Aim to refactor things only the things that you have to, and the things that you understand. **Keep the stuff that works, and don't change too much.** Test harnesses keep us safe – when we are making large changes we can ensure that we did not break the system by unit testing at the micro level, and integration testing at the macro level. If a Legacy System doesn't have tests then we'll have to add them – but this code might not be amenable to testing. In order to be able to effectively test particular units of a system, we need to be able to break dependencies so that during the test phase we can test an isolated unit. During our unit test, we do not want the code to be writing data into a database. We can break these dependencies by making use a **seam. A** seam is a place where you can alter the behaviour of your program without editing it in that place. This is the point where you decide to use one behaviour or the other. To make use of an object seam, we must be able to pass in our test implementation, instead of the real implementation of the dependency. This may require a refactor to do so. We may also want our special test code implementation to check other things: e.g. if we have some code that writes its results into a database, check if it works in our test implementation

19) Code Metrics:

Coupling is one metric. **Stability:** ratio of afferent to efferent couplings of an object. Objects (at the core of the system) that are depended on a lot shouldn't be changed much, if an object is changed a lot we would hope it is one on the edge of the system. **Dependencies** can be represented by a **dependency structure matrix** which are useful as they help us find cyclic dependencies which indicate **very tight coupling**. **Complexity** of code can be computed in a few ways: 1) **McCabe Complexity** (treats prog as CFG, counts nodes and edges to try work out number of exec. paths). Gives lower bound of unit tests needed for the program too. 2) **WILT** – we compute the integral of the indented area. 3) **ABC Metric** – counts the occurrences of Assignments, Branches, Conditional statements. Function calls get counted too (Flog). Flay is a tool which looks for cloned code by examining the AST. A file's **turbulence** – the number of commits which changed it. We can also measure what files get changed in the same commits to see what things are coupled.

20) The Waterfall Development Model:

The Waterfall Development model sets out a number of phases (e.g. System Requirements, Software Requirements, Analysis, Design, Implementation, Testing, Deployment, Maintenance) for project development. The idea is that once the previous phase is completed, we are completely done with that task and do not rework it. This issue is that this model is too rigid - and we can't fix previous parts that might be wrong / doesn't allow much space for client feedback along the way.

20.1) The Agile Development Model:

Teams work in short, iterative development cycles called "sprints," which typically last one to four weeks. At the end of each sprint, the team delivers a potentially shippable product. The first release happens much more quickly, so we can get feedback and generate revenue more quickly. Agile teams use regular, frequent communication and collaboration with customers to gather feedback and adjust their plans as needed. Agile methodologies are designed to be flexible and adaptable, and can be applied to a wide range of software development projects. They are particularly well-suited to projects with rapidly changing or evolving requirements, or where the requirements are not fully understood at the outset of the project.

20.1.1) Scrum:

Scrum is a framework for agile software development. It is a **lightweight, iterative, and incremental approach to software development that emphasises collaboration, flexibility, and the ability to respond to change**. A project is divided into small, iterative development cycles called "sprints," which typically last one to four weeks. At the beginning of each sprint, the team selects a set of high-priority features to be implemented in the current sprint. The team then works together to develop and deliver the selected features by the end of the sprint. Scrum teams consist of a **product owner, a Scrum master, and the development team**. The Product Owner is responsible for defining and prioritising the features to be developed. The Scrum master is responsible for ensuring that the Scrum process is followed, and the development team is responsible for actually developing the software. Scrum teams use regular, frequent communication and collaboration to ensure that the project is on track and to gather feedback from customers and stakeholders**. This is done through regular meetings called "scrums,"** which are short, focused discussions to review progress and identify any obstacles that need to be addressed. Scrum masters - facilitate the progress. Product owner - decisions on what the clients get.

20.1.1) Extreme Programming

This is a software development methodology that emphasises the importance of customer satisfaction, communication, and simplicity. It was developed as a response to problems seen in traditional software development processes. XP is based on the following principles:

Communication: The development team should communicate constantly and openly with the customer, and **all team members should be involved in the decision-making process**.

Simplicity: The development team should focus on the most important features of the software and keep the **design as simple as possible**.

Feedback: The development team should **get frequent feedback from the customer** and use it to make adjustments to the software.

Courage: The development team should have the **courage to make changes to the software when necessary**, even if it means throwing away code that has already been written.

XP involves a number of specific practices, such as pair programming, TDD (write tests before code), and CI (where code changes are integrated into the main branch of the codebase frequently).

XP is particularly well-suited for projects that require rapid iteration and frequent changes.

20.1.2) Kanban

Kanban is based on the idea of using a "kanban board" **to track the progress of work** through a series of columns, each representing a different stage of the process.

Work is represented by cards or items on the kanban board, and the **board is divided into columns that represent the different stages of the workflow**. For example, a kanban board for a software development project might have columns for "To Do", "In Progress", and "Done". As work is completed, items are moved from one column to the next, allowing team members to see the progress of the work at a glance.

One of the key principles of Kanban **is the idea of "pull" rather than "push".** In a traditional "push" system, work is pushed through the workflow as quickly as possible, regardless of whether it is actually needed. In a Kanban system, work is only **pulled into the next stage of the workflow when it is needed, which helps to prevent overloading and waste**.

Kanban is particularly well-suited for projects that involve frequent changes and the need for flexible scheduling. It can be used in a variety of different contexts, including manufacturing, product development, and service delivery.

We must focus on what is most important to do right now, we don't get ahead of ourselves. We just do what is next needed for a more useful product.

20.1.3) Continuous Integration

To be Agile, we want to continually push our small changes to the master branch frequently with short lived branches. To do this effectively we need an automated build which will compile, run all unit tests, restyle, do checks, and then package for release. Without CI, we could make mistakes doing these things. CI Servers such as Jenkins can monitor pushes to server control, checkout the code, clean everything, run all the tests and build the application. It can also gather statistics on e.g. test failures, what changes have been made. It then tells us whether we are ready to release or not and if our feature is complete.

21) Tell, Don't Ask

"Tell, don't ask" means **classes should tell other classes to do something rather than asking for the information to do it themselves.** This is good as commanding classes reduces coupling in code – using commands means that fewer classes have to know about the internals. If any internals then get changed, many classes will have to be changed too.

Something like "`boxOffice.getCustomerDatabase().getCustomer(customer).getTickets().add(ticketId);`" is known as a **trainwreck** and shouldn't be around if the Law of Demeter is followed.

22) Identifying Design Patterns / Design Patterns to use:

1) Template Method Pattern: If we have some methods composed to do a process in some superclass, and subclasses override some of these methods then we have this pattern. If we are in a situation where there is lots of code duplication of the same methods, then we can refactor out the common code to some superclass and leave the differing code in the subclasses.

2) Strategy Pattern: If we have some field which is an interface type, and we call into a method of the field to select our "execution strategy" then we have this pattern. If we are in a situation where we want to select which method to execute based on what class we are in then we should use this pattern.

3) Factory Pattern: If we have some method which instantiates some classes based on some input data at runtime then we have a Factory. If we want more convenient instantiation of objects based on some data we have then we should use this pattern.

4) Builder Pattern: If we are sequentially composing methods to define fields of some complex class then we are using this pattern. If we are trying to define some complex constructor then we should use this pattern to initialise each field step by step.

5) Singleton Pattern: If we have an object that has a private constructor and some other method which only instantiates the object once / refers to some previously defined version of this object then we are using the Singleton Pattern. If we are confident that only one version of a class is required and want global access to it (and want to avoid instantiating it multiple times – could be expensive) then we should use this pattern. We should also ensure that we can test properly (do we have a seam or are we forced to always use the singleton?)

6) Command Pattern: If we're submitting commands to some queue and processing them – typically in a multi threaded fashion then we're using this pattern. If we want to do load balancing and have some threads dedicated to producing work and others carrying out then this pattern is good.

7) Observer Pattern: This is present if we have some sort object that gets updated as a result of specific changes to our current object (usually stored a field – could be a list of observers). We should use this pattern if we want other objects to respond to changes to our current object.

8) MVC Pattern: If we have some sort of Model which is completely isolated, except for the fact that it has a View observer, a controller which has reference to this model and calls its methods, and a view which has the controller as an observer then we are using this pattern. We should use this pattern to structure GUI code.

9) PAC Pattern: If we have agents that only know about agents (which only control a specific part of the UI) above and below them. We can use this pattern when structuring GUI code.

10) The Adaptor Pattern: Related to Hexagonal Architecture. It is in use if we see some sort of adapter class which implements an interface which is in use in our program, and holds an inner class (usually a third party library class) to which it delegates most method calls to, while also implementing the methods of the interface it has. **We should use it when we want to insulate our implementation from a third party library / something which may change unexpectedly.**

11) Tell; Don't Ask

If our classes tell other classes to perform a specific task, rather than seeking specific information from those classes to then carry out the operation themselves in their own method call then we are following this pattern. We would want to follow this pattern so that we don't have trainwrecks ("`boxOffice.getCustomerDatabase().getCustomer(customer).getTickets().add(ticketId);`") so we can avoid violating the Law of Demeter (and thus following this reduces the coupling of our code, and means that classes don't need to know about the internals of other far off classes, thereby making our code less rigid, immobile and fragile).

12) Hexagonal Architecture:

If we have internal logic that is insulated from the outside (3rd party libraries or other objects that interact with outside systems) through the use of ports and adapters then we are using this architecture. If we want to insulate our inner business logic from 3rd party libraries then we should setup a hexagonal architecture.

13) The Decorator Pattern:

The Decorator Pattern is in use if we have some class A as a field of another class B, and B adds some extra methods, or does some processes before and/or after deferring to methods in A. We should use it if we want to add more features to a class without subclassing.

The following patterns are less likely to be the subject of an exam question

14) The Facade Pattern:

The Façade pattern is in use if we have an interface in front of a more complicated API to simplify the use of the complicated API / tailor it to a specific use. The facade pattern is useful when you want to **provide a simple interface to a complex system**, or when you want to **decouple a client from the implementation details of a subsystem**.

15) The Simplicator Pattern:

Similar to the façade pattern – it is in use when we provide a simpler easier to use API in front of a hard to use class.

16) The Proxy Pattern:

This is in use when we are controlling access to some object and its methods in some way – we have some class in front of it. This is useful if we want to reduce access to our other class. Implement security clearance or cache results.

16) Refactoring Techniques:

1) Compose Method – break down long methods into submethods. Gives names to pieces of code, improves abstract ion and clarity. **2) Inlining Variables** – If we have variables that are used only once, we can just put their definition in place of where they were used and get rid of the variable. **3) Extracting to Class or Method** – If we have duplication between classes we can extract it out to a common class and then call that classes' method. Similarly with methods. **4) Using Polymorphism to replace Conditionals** – If we call a method conditionally, rather than passing a Boolean into somewhere else which then checks which method to call, we should just make some interface which two different classes implement with an overridden method. We just call the method then, and polymorphism handles all the checking