SOFTWARE ENGINEERING DESIGN

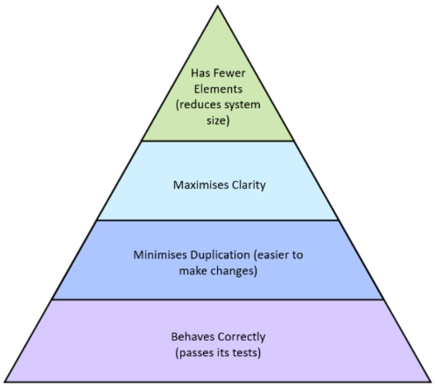
Waterfall development: Requirement gathering → Analysis → Design → Coding → Testing → Deployment

Bad as not much scope for going back and reworking things after a phase ends.

Not used very much if at all today.

Cost of change: with bad design the cost of making changes is large and you end up with a "Ball of Mud". Keep design simple and improving design over time allows for effective evolution and maintenance of the software.

Four Elements of Simple Design Pyramid →


Has Fewer Elements (reduces system size)
Maximises Clarity
Minimises Duplication (easier to make changes)
Behaves Correctly (passes its tests)

TEST DRIVEN DEVELOPMENT

Cycle: Write a failing test → Write enough code to pass the test → Refactor → Repeat

Or: API Design (specifying how things should work) → Internals Design (initial implementation to achieve working code) → Structural Design (refactoring the design to avoid Ball of Mud, with safety net of tests to avoid breaking functionality) → Repeat

BDD – Behaviour Driven Development: Write a few behavioural properties of the object we want to create (an informal specification) → Translate requirement into unit tests (JUnit tests: public class, @Test public void [sentence](), assertTrue(x), assertFalse(x), assertThat(x, is(y))) → Write code to pass → Refactor → Move to next requirement in informal specification

REFACTORING

Only refactor in TDD when in a green state (all tests passing). Refactoring should be applied little and often.

Technical debt: leaving refactoring of badly written code/features until later.

Compose Method: breaking down a long method to make it shorter (extract parts out into their own functions with appropriate names). Improves abstraction.

Separating Responsibilities: for example, changing a for loop with 2 functions in the body to 2 for loops which are responsible for one piece of functionality each (allows for functionality to be more easily abstracted out).

In-lining Variables: removing the use of a new variable to store a value when it is not required (the function/value can simply be put as is where it needs to be used).

Removing Duplication Between Classes: first refactor similar things to be the same, then refactor away these commonalities (perhaps into another object/class – more on this later!)

Renaming Methods: sometimes method names can be unclear so renaming them aids readability.

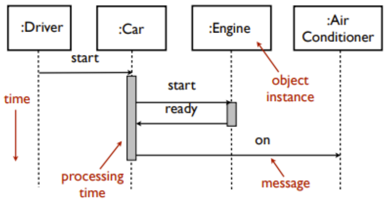
Replace Conditional with Polymorphism: we try to avoid conditional statements based on information queried from a collaborator – instead we want the collaborator to make the decision without us knowing.

MOCK OBJECTS

Mock objects help to test the communication between objects in our systems. The interactions between objects in OOP is a very important aspect of the design of the system.

OOP is a web of objects where each object can send/receive messages in the form of function calls, informing the objects of some new state of the world, and expect that they will do something appropriate in response.

UML Sequence Diagrams:



Command: tell another object to do something for us (delegate responsibility), do not usually get return value, often change the state of the invoked object or other part of program.

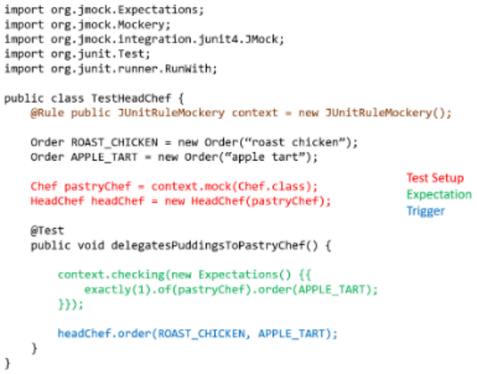
Query: ask another object to tell us about a value for our use. Queries return a value but should not have side-effects on the state of the invoked object.

Value Objects: these objects represent values and are often the leaves of the object chart. Testing these objects is often done using a state-based approach, checking for values of the object at certain points in the testing execution.

Tell Don't Ask: the only calls which should return values are queries and these should not change any state or interaction between objects. Commands should not expect a return value from calls.

Focus on a Single Object: when writing unit tests we focus on a single object at a time, testing its interactions with other objects as opposed to its internal state (hence, the collaborators play a key role in the tests).

Collaborate Through Roles: Java uses Interfaces to represent roles, i.e., what the purpose of an object is. The calling object sees it's collaborator as having a particular role (and good OO design allows for objects to be interchanged, if they implement the relevant Interface). This helps test one object without needing to have implemented its collaborators.



```
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.junit.Test;
import org.junit.runner.RunWith;

public class TestHeadChef {
    @Rule public JUnitRuleMockery context = new JUnitRuleMockery();

    Order ROAST_CHICKEN = new Order("roast chicken");
    Order APPLE_TART = new Order("apple tart");

    Chef pastryChef = context.mock(Chef.class);
    HeadChef headChef = new HeadChef(pastryChef);

    @Test
    public void delegatesPuddingsToPastryChef() {

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

        headChef.order(ROAST_CHICKEN, APPLE_TART);
    }
}
```

We trigger inward arrows on our object dependency graph in the tests by making calls to the object being tested, and we use Mock Objects (from the jMock2 library) to test the object making calls to its collaborators.

DESIGNING FOR FLEXIBILITY

Bad Design Characteristics: Rigidity = code is hard to understand or easily change, Fragility = when changing one part, another changes unexpectedly, and Immobility = hard to reuse elements of the code in other applications.

Encapsulation: battles fragility. If the implementation of an object is encapsulated, changing the way it works internally should not affect the rest of the code which makes use of it.

Information Hiding: this concerns abstraction. Helps with chunking the program into concepts we can understand.

4 visibility levels in Java: public (any object can directly access the variable/method), protected (subclass objects, and members of classes in the same package can access), package/default (classes within the same package can access), and private (denies all access to objects not in the same class).

public/protected = part of the class's API (others can use this, and we cannot make it private later without risking breaking code others have written)

Internal state should be private to an object, even for testing purposes.

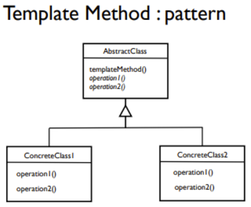
Using Tell Don't Ask leads to a more flexible design which is easier to evolve and change.

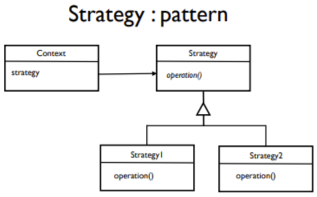
Train wreck: getX().getY().getZ().doSomething() (many queries for one command)

Law of Demeter: 1. Each unit should only have limited knowledge about other units, 2. Each unit should only talk to its friends and not strangers, and 3. Only talk to your immediate friends.

Good encapsulation reduces the "blast radius" when you need to change a piece of code in one object.

RE-USE AND EXTENSIBILITY





Template Method: extract commonality to a superclass (abstract class) then inherits from that class, implementing small implementation specific changes (implement the abstract method(s)).

Strategy: avoids inheritance and favours composition by delegating common code to a peer object which collaborates with other objects.

Open-Closed Principle: a class's behaviour should be extensible without modification.

Separate the things that stay the same from those which change. Change behaviour by adding new code, not changing existing (working) code.

Coupling: close coupling (e.g., inheritance) causes immobility, for example in the Template Method if we want to use one of the subclasses, we need to bring the superclass (abstract class) along with it.

Coupling Metrics: Afferent coupling (Ca) = measures the number of other classes using this class/module – measures the class's responsibility. Efferent coupling (Ce) = measures the number of classes this class makes use of – measures the class's independence.

CODE METRICS

Stability: the coupling metrics given above can be used to measure the stability of an object (a balance of its independence and responsibility). We want objects at the core of the system which are depended on by many other components to be stable, and those unstable objects (ones which change frequently) to exist at the edge of a system, with few things depending on them.

Dependency Structure Matrix: helps compare the Ca and Ce for different modules and to detect cycles in the dependency graph (cycles = tight coupling and immobility). These can be generated by software such as NDepend.

McCabe Complexity: gives a lower bound for the number of tests required for a unit, by counting the numbers of nodes and edges in the control flow graph for the program and counting the possible different executions that could happen. This is expensive to calculate.

WILT: Whitespace Integrated over Lines of Text for a given piece of code. By integrating over the indented area, we get a measure of complexity. There is a strong correlation between WILT and McCabe complexity. This is a lot cheaper to calculate than McCabe.

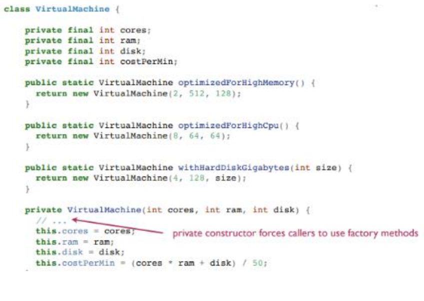
ABC Metrics: count the number of Assignments, Branches, and Conditions as a measure of complexity. Example: Flog for Ruby. Flay for Ruby detects code duplication.

Lifelines: plot the complexity of code over time.

Turbulence: plotting the number of commits made to each file against their complexity → there are 4 quadrants – 1. Simple and rarely change, 2. Complex and rarely change, 3. Simple and change often, and 4. Complex and change often.

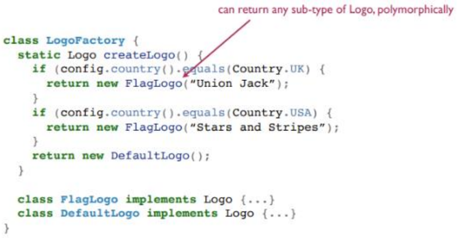
Temporal Coupling: when things change at the same time. This can be seen when tracking which files are generally changed in the same commit (something which may not be obvious from the static view of the code).

CREATING OBJECTS AND MANAGING DEPENDENCIES



```
class VirtualMachine {

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

    public static VirtualMachine optimizedForHighMemory() {
      return new VirtualMachine(2, 512, 128);
    }

    public static VirtualMachine optimizedForHighCpu() {
      return new VirtualMachine(8, 64, 64);
    }

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

    private VirtualMachine(int cores, int ram, int disk) {
      // ...
      this.cores = cores;
      this.ram = ram;
      this.disk = disk;
      this.costPerMin = (cores * ram + disk) / 50;
```

Factory Method: a method whose job it is to create objects. These help to increase clarity and flexibility around object creation. They can also be used to defer choice of which class parameters to use until runtime. These are named static methods on a class that internally calls the constructor and returns the object. We can make the constructor for the class private to ensure the usage of the factory methods for a class.

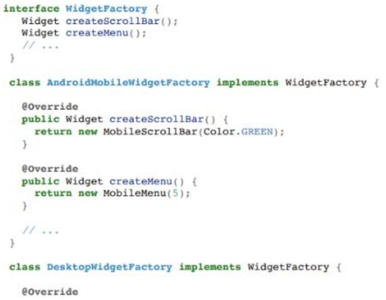


can return any sub-type of Logo, polymorphically

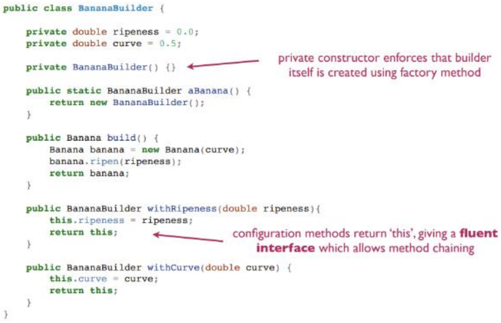
```
class LogoFactory {
  static Logo createLogo() {
    if (config.country().equals(Country.UK) {
      return new FlagLogo("Union Jack");
    }
    if (config.country().equals(Country.USA) {
      return new FlagLogo("Stars and Stripes");
    }
    return new DefaultLogo();
  }

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

Factory Object: can be used to defer the decision about the exact type of the object to be created until runtime by using some properties which are only calculated at runtime.

Abstract Factory Pattern: when we have a family of different but related object to create, and the exact types of these objects depend on the same condition, we can create a type hierarchy including different types of factory, then use this factory wherever we need to create a new object. Sometimes a factory can be used to instantiate the right factory.



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

 class AndroidMobileWidgetFactory implements WidgetFactory {

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

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

   // ...
 }

 class DesktopWidgetFactory implements WidgetFactory {

   @Override
```



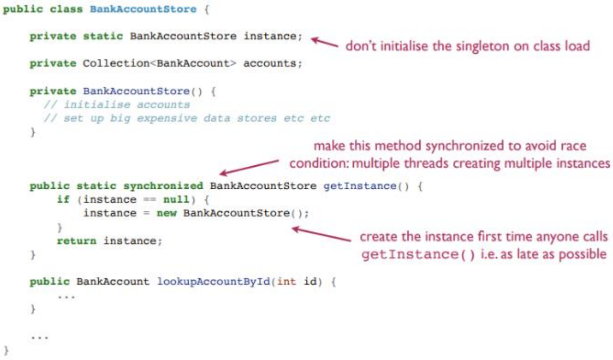
```
public class BananaBuilder {

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

    private BananaBuilder() {}

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

    public Banana build() {
        Banana banana = new Banana(curve);
        banana.ripen(ripeness);
        return banana;
    }

    public BananaBuilder withRipeness(double ripeness){
        this.ripeness = ripeness;
        return this;
    }

    public BananaBuilder withCurve(double curve) {
        this.curve = curve;
        return this;
    }
}
```

private constructor enforces that builder itself is created using factory method

configuration methods return 'this', giving a **fluent interface** which allows method chaining

Builder: this is an alternative to the factory. A builder is an object separate from the one being created that takes responsibility for gathering configuration parameters, then using those to produce a fully formed object in a valid state. [PIZZA EXAMPLE]. Constructor returns a builder, then this has various with() methods on it for each parameter, and finally calling build() returns the object, with default values where a with() method has not been used.

Singleton: the aim of this pattern is to ensure that we only have one instance of a particular object in our system and provide a global point of access to it, for example where the object takes a long time to instantiate. This should be used sparingly as global variables introduce more dependencies (coupling) and make the code hard to test and reuse. This can be implemented by having a class which guards the instance, providing a getInstance() method and having a private constructor for this singleton. The code shows how to lazily instantiate the singleton (i.e., only when it is required) and how to ensure this is thread-safe to avoid multiple instances of the singleton appearing.



```
public class BankAccountStore {

    private static BankAccountStore instance;

    private Collection<BankAccount> accounts;

    private BankAccountStore() {
      // initialise accounts
      // set up big expensive data stores etc etc
    }


    public static synchronized BankAccountStore getInstance() {
        if (instance == null) {
            instance = new BankAccountStore();
        }
        return instance;
    }

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

    ...
}
```

don't initialise the singleton on class load

make this method synchronized to avoid race condition: multiple threads creating multiple instances

create the instance first time anyone calls getInstance() i.e. as late as possible

The dependency created by the singleton pattern can be avoided by simply passing the singleton in. Using an interface to specify the behaviour of this object means any object implementing the interface can be passed in. This reduces the direct dependency on the singleton being created.

WORKING WITH LEGACY SOFTWARE

Legacy System: a piece of software you have inherited and that is of value to you.

We first want to figure out the dependencies which exist in the system to try and understand it. These may be linking dependencies (e.g., one class referring to another/third party library) or dependencies on external services (e.g., databases/other services communicating with our system). We can visualise these in a graph.

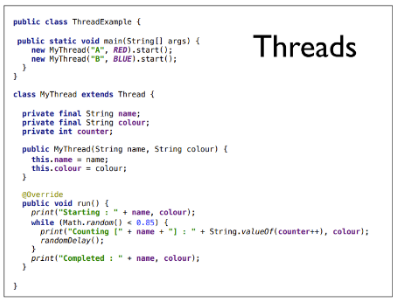
Ensure any changes made preserve correct behaviour – changing code you do not understand can be risky. We can ensure correct behaviour is maintained by using automated testing. We unit test at the micro level and system test at the macro level.

Seams: to effectively test units of a system, we need to be able to break dependencies to test an isolated unit of the system. We can break dependencies on writing to databases or sending emails by using seams. Seams are places where you can alter the behaviour of your program without editing it in that place.

Enabling Point: the point where you decide to use one behaviour or the other. This is the point at which we pass in our test implementation as opposed to the real implementation of the dependency. This may not be possible if the code being tested uses the new operation to create its dependency or refers to a singleton instance. We may in this case need to refactor so that a reference to the dependency can instead be passed in from outside.

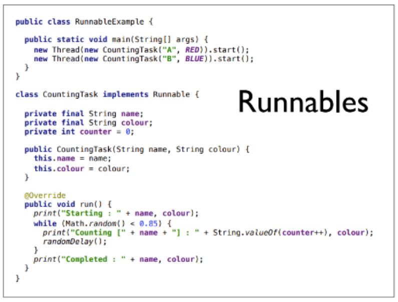
CONCURRENCY PATTERNS

Threads: the most common way of doing concurrency in Java. We write a class which extends the Thread class, overriding the run() method and calling start() on the thread. This introduces a close coupling due to the inheritance.



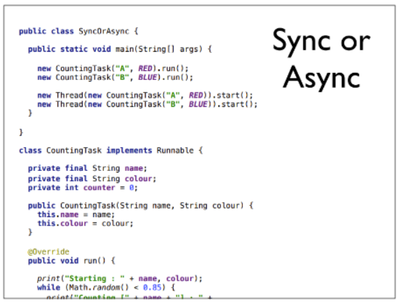
Threads

```
public class ThreadExample {

 public static void main(String[] args) {
   new MyThread("A", RED).start();
   new MyThread("B", BLUE).start();
  }
}

class MyThread extends Thread {

  private final String name;
  private final String colour;
  private int counter;

  public MyThread(String name, String colour) {
    this.name = name;
    this.colour = colour;
  }

  @Override
  public void run() {
    print("Starting : " + name, colour);
    while (Math.random() < 0.85) {
      print("Counting [" + name + "] : " + String.valueOf(counter++), colour);
      randomDelay();
    }
    print("Completed : " + name, colour);
  }

}
```



Runnables

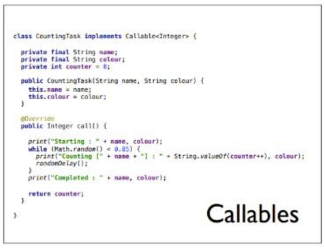
```
public class RunnableExample {

  public static void main(String[] args) {
    new Thread(new CountingTask("A", RED)).start();
    new Thread(new CountingTask("B", BLUE)).start();
  }
}

class CountingTask implements Runnable {

  private final String name;
  private final String colour;
  private int counter = 0;

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

  @Override
  public void run() {
    print("Starting : " + name, colour);
    while (Math.random() < 0.85) {
      print("Counting [" + name + "] : " + String.valueOf(counter++), colour);
      randomDelay();
    }
    print("Completed : " + name, colour);
  }
}
```



Sync or Async

```
public class SyncOrAsync {

  public static void main(String[] args) {

    new CountingTask("A", RED).run();
    new CountingTask("B", BLUE).run();

    new Thread(new CountingTask("A", RED)).start();
    new Thread(new CountingTask("B", BLUE)).start();
  }

}

class CountingTask implements Runnable {

  private final String name;
  private final String colour;
  private int counter = 0;

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

  @Override
  public void run() {

    print("Starting : " + name, colour);
    while (Math.random() < 0.85) {
      print("Counting [" + name + "] : " +
```

Runnables: we can break the inheritance in Threads by composing objects following the Strategy pattern. We create a runnable (an object that implements the run() method) which can be passed to a thread's constructor. Runnables encapsulate some behaviours to be run at a particular time – they can be run synchronously and asynchronously. Runnables can be used without a Thread by calling their run() method directly.

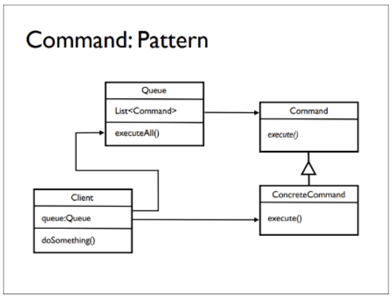


```
class CountingTask implements Callable<Integer> {

  private final String name;
  private final String colour;
  private int counter = 0;

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

  @Override
  public Integer call() {

    print("Starting : " + name, colour);
    while (Math.random() < 0.85) {
      print("Counting [" + name + "] : " + String.valueOf(counter++), colour);
      randomDelay();
    }
    print("Completed : " + name, colour);

    return counter;
  }

}
```

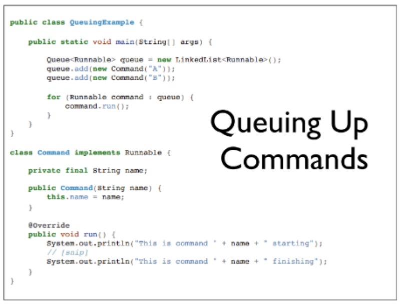
Callables

Callables: an extension of this idea introduced in more recent versions of the JDK. They are like runnables but allow values to be returned from the call() method and allow exceptions to be thrown if necessary.

Runnables and Callables are both examples of a device where a piece of behaviour can be wrapped up and passed around and executed in different contexts. Commands can be appended to a queue or log so they can be executed in a batch or keep a history of past commands. The abstract Command design pattern shown here has a client which creates commands and adds it to a queue of commands which can later be executed as a batch.



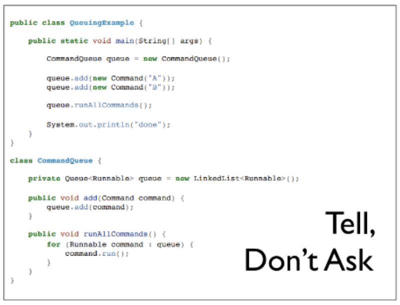
Command: Pattern



```
public class QueuingExample {

    public static void main(String[] args) {

        Queue<Runnable> queue = new LinkedList<Runnable>();
        queue.add(new Command("A"));
        queue.add(new Command("B"));

        for (Runnable command : queue) {
            command.run();
        }
    }
}

class Command implements Runnable {

    private final String name;

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

    @Override
    public void run() {
        System.out.println("This is command " + name + " starting");
        // [snip]
        System.out.println("This is command " + name + " finishing");
    }
}
```

Queuing Up Commands

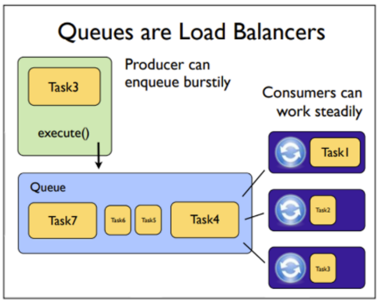
Queue: to make a queue of commands, we can either make use of a Queue<Runnable> and create a concrete Command class, add commands to this Queue then iterate over it, calling run() on each item in the queue. Alternatively, if we want to use a Tell, Don't Ask style, we can implement a class which can handle the queue, without leaking abstraction.



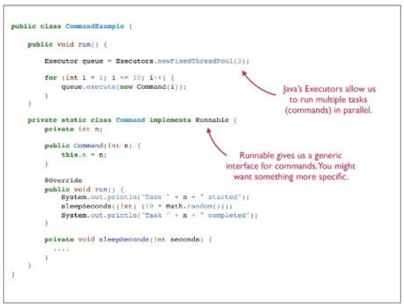
```
public class QueuingExample {

    public static void main(String[] args) {

        CommandQueue queue = new CommandQueue();

        queue.add(new Command("A"));
        queue.add(new Command("B"));

        queue.runAllCommands();

        System.out.println("done");
    }
}

class CommandQueue {

    private Queue<Runnable> queue = new LinkedList<Runnable>();

    public void add(Command command) {
        queue.add(command);
    }

    public void runAllCommands() {
        for (Runnable command : queue) {
            command.run();
        }
    }
}
```

Tell, Don't Ask

Executor: executors (part of the java.util.concurrent package) can manage a pool of threads and use these to execute enqueued tasks concurrently. Producers call execute() on tasks which enqueues them in the executor, waiting to be sent to Consumers (threads in the thread pool) to be executed. Queues are load balanceers – they can work steadily and ensure short running tasks are executed in a timely fashion while longer tasks are also executed.



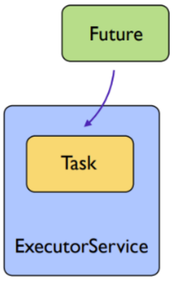
Queues are Load Balancers



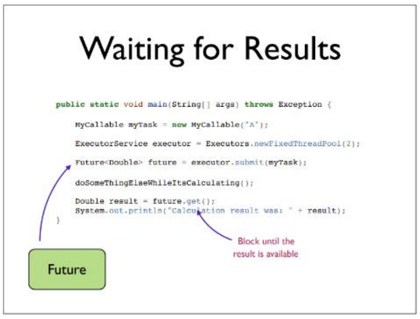
```
public class CommandExample {

    public void run() {

        Executor queue = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 10; i++) {
            queue.execute(new Command(i));
        }
    }

    private static class Command implements Runnable {
        private int n;

        public Command(int n) {
            this.n = n;
        }

        @Override
        public void run() {
            System.out.println("Task " + n + " started");
            sleepSeconds((int) (10 * Math.random()));
            System.out.println("Task " + n + " completed");
        }

        private void sleepSeconds(int seconds) {
          ....
        }
    }
}
```

Java's Executors allow us to run multiple tasks (commands) in parallel.

Runnable gives us a generic interface for commands. You might want something more specific.



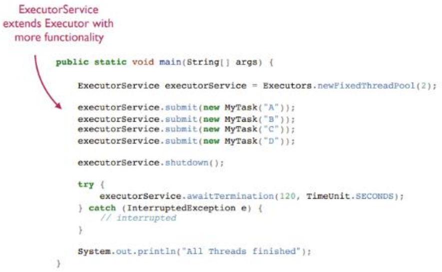
Futures: when submitting a task to an executor using a Callable, it returns a Future. Futures can be used to track the execution of our task. There are many different methods on Futures such as get(), get(timeout), cancel(mayInterruptIfRunning), isDone(), and isCancelled() to name a few.



Waiting for Results

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

    MyCallable myTask = new MyCallable("A");

    ExecutorService executor = Executors.newFixedThreadPool(2);

    Future<Double> future = executor.submit(myTask);

    doSomeThingElseWhileItsCalculating();

    Double result = future.get();
    System.out.println("Calculation result was: " + result);
}
```

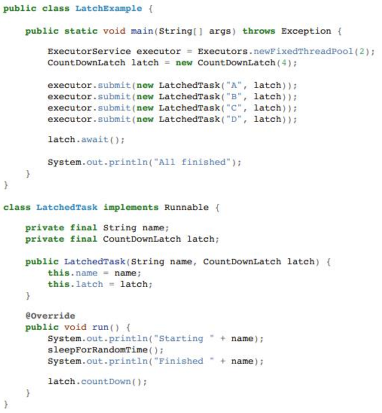
Block until the result is available



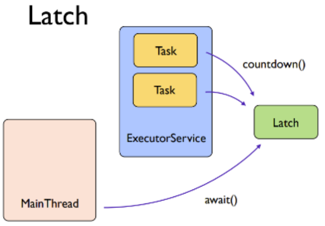
ExecutorService extends Executor with more functionality

```
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"));

    executorService.shutdown();

    try {
        executorService.awaitTermination(120, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        // interrupted
    }

    System.out.println("All Threads finished");
}
```

ExecutorService: these provide more functionality than a basic Executor. They can allow us to wait for all tasks to complete and, for example, inform the user of this or quit the application when everything is done. In the code here, after submitting all tasks, we tell the executor to shutdown, and then wait for termination. Once this call returns, we know the executor's queue is empty (or the timeout has expired).



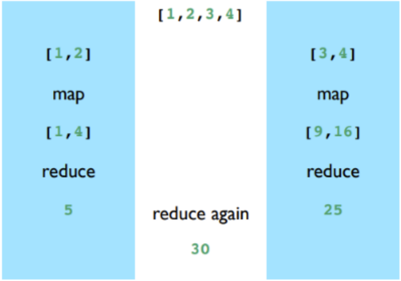
```
public class LatchExample {

    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));

        latch.await();

        System.out.println("All finished");
    }
}

class LatchedTask implements Runnable {

    private final String name;
    private final CountDownLatch latch;

    public LatchedTask(String name, CountDownLatch latch) {
        this.name = name;
        this.latch = latch;
    }

    @Override
    public void run() {
        System.out.println("Starting " + name);
        sleepForRandomTime();
        System.out.println("Finished " + name);

        latch.countDown();
    }
}
```



Latch: alternative ways of doing this include latches, in cases where we may not want to shut down the ExecutorService because we want to use it again for the next set of tasks, but we still want to know when all of the tasks in one group are complete in order to proceed. We create the latch in the main thread, with a counter set to the number of tasks to do. We pass the latch to each task, and at the end of the task they make the latch count down. In the main thread we wait for each latch to count to reach 0. The code to the left shows how this works in practice.

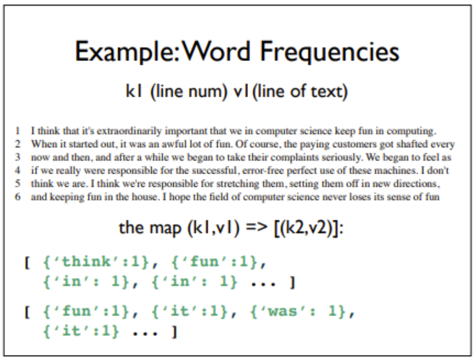
MAPREDUCE

MapReduce is a system that Google built to work with very large sets of data. Mapping and reducing are not new – they have been in functional programming for a while now – however the idea of MapReduce has become more popular in SWE more recently, inspiring other implementations such as Hadoop (which also uses the map-reduce technique).



MapReduce was created because Google wanted to index the entire world wide web which is a very large amount of data. MapReduce allows for these large computations to take place over multiple machines in parallel – the computation was divided into two parts: 1. Applying a map function, and 2. Reducing over the results to aggregate them. This example shows how the sum of the squares of all values in a list can be executed in parallel using MapReduce. Here, map = squaring each value, and reduce = summing these values.

The actual implementation of MapReduce is a little more complex, as it works with key-value pairs, however this makes it more powerful and flexible.



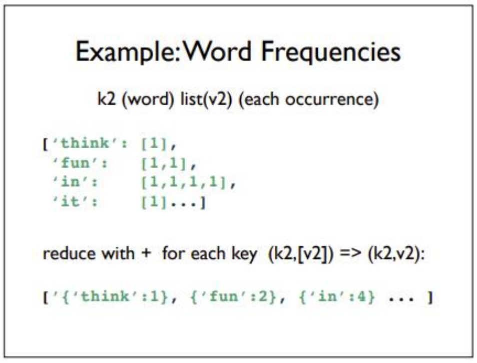
Example: Word Frequencies

k1 (line num) v1 (line of text)

1 I think that it's extraordinarily important that we in computer science keep fun in computing.
2 When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every
3 now and then, and after a while we began to take their complaints seriously. We began to feel as
4 if we really were responsible for the successful, error-free perfect use of these machines. I don't
5 think we are. I think we're responsible for stretching them, setting them off in new directions,
6 and keeping fun in the house. I hope the field of computer science never loses its sense of fun

the map (k1,v1) => [(k2,v2)]:

```
[ {'think':1}, {'fun':1},
  {'in': 1}, {'in': 1} ... ]

[ {'fun':1}, {'it':1}, {'was': 1},
  {'it':1} ... ]
```



Example: Word Frequencies

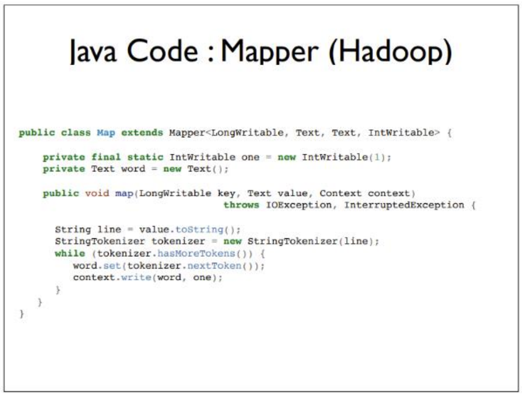
k2 (word) list(v2) (each occurrence)

```
['think':  [1],
 'fun':    [1,1],
 'in':     [1,1,1,1],
 'it':     [1]...]
```

reduce with + for each key (k2,[v2]) => (k2,v2):

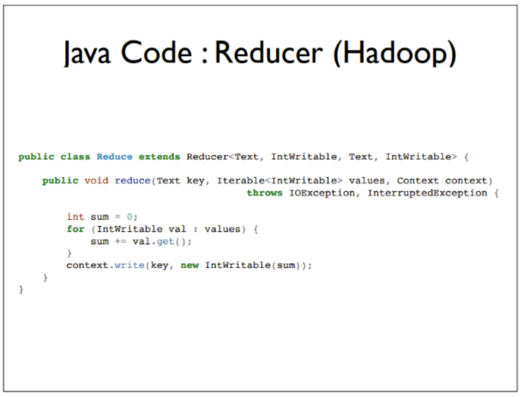
```
['{'think':1}, {'fun':2}, {'in':4} ... ]
```

Hadoop: Google’s MapReduce inspired other implementations of MapReduce to allow for large datasets to be processed quickly using many machines – one of the most widely used is Hadoop. Note Google’s MapReduce is not open source.



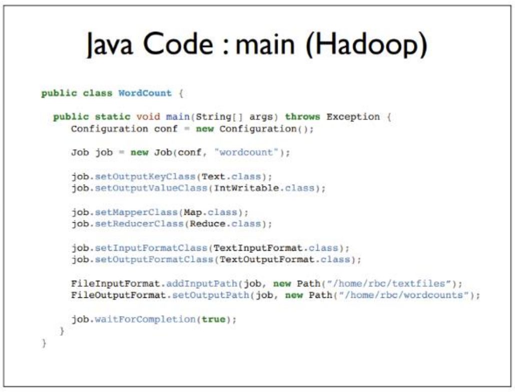
Java Code : Mapper (Hadoop)

```
public class Map extends Mapper<LongWritable, Text, Text, IntWritable> {

    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);
      }
   }
}
```



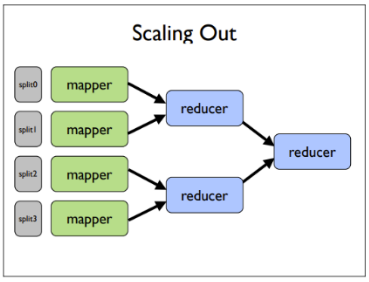
Java Code : Reducer (Hadoop)

```
public class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
                                                throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
}
```



Java Code : main (Hadoop)

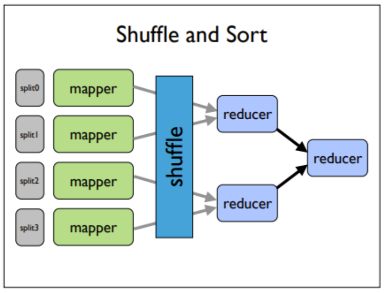
```
public class WordCount {

  public static void main(String[] args) throws Exception {
     Configuration conf = new Configuration();

     Job job = new Job(conf, "wordcount");

     job.setOutputKeyClass(Text.class);
     job.setOutputValueClass(IntWritable.class);

     job.setMapperClass(Map.class);
     job.setReducerClass(Reduce.class);

     job.setInputFormatClass(TextInputFormat.class);
     job.setOutputFormatClass(TextOutputFormat.class);

     FileInputFormat.addInputPath(job, new Path("/home/rbc/textfiles"));
     FileOutputFormat.setOutputPath(job, new Path("/home/rbc/wordcounts"));

     job.waitForCompletion(true);
   }
}
```



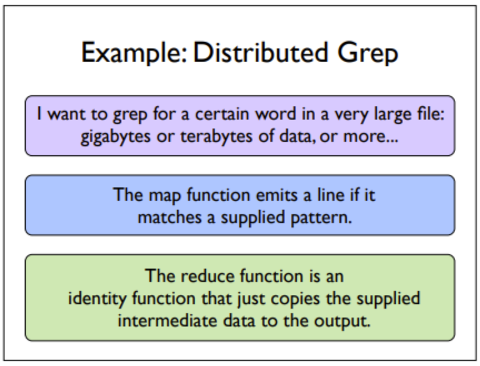
Scaling Out

When MapReduce runs it splits the input data into several “splits” with a separate mapper, most likely running on a separate computer. The outputs are then run through the reducer phase. There are normally fewer reducers than mappers depending on the complexity of various parts of the job. Reducers can be chained if the reduce function is commutative and associative. This can lead to improved efficiency as initial reductions can be performed nearer to mappers, reducing the amount of data which must be moved around.

The magic of MapReduce is in the shuffle. This is where all 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.



Shuffle and Sort

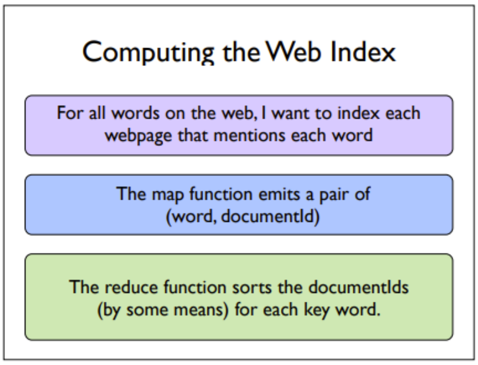


Example: Distributed Grep

I want to grep for a certain word in a very large file: gigabytes or terabytes of data, or more...

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.



Computing the Web Index

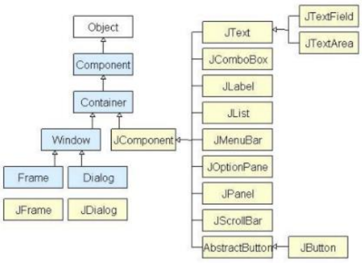
For all words on the web, I want to index each webpage that mentions each word

The map function emits a pair of (word, documentId)

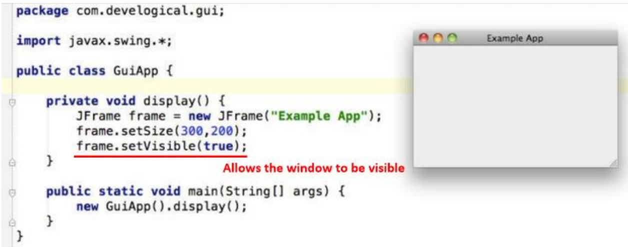
The reduce function sorts the documentIds (by some means) for each key word.

INTERACTIVE APPLICATIONS

Interactive applications are those which have GUIs and which need to respond to people clicking on widgets, and to update the display in response to input and changes.

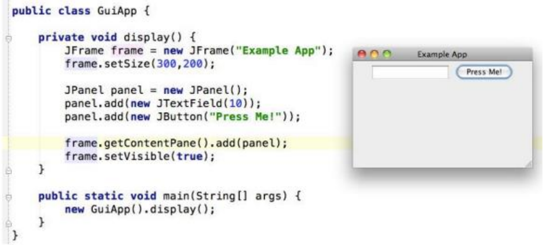


We often use a library of components to put together a UI. Here is an example using the Java Swing GUI library. It comprises a collection of different types of widget classes that can be assembled to display windows, menus, and dialog boxes. Note that we prefer to just use a JFrame as opposed to creating a class to extend the JFrame (as many people do, creating close coupling). Layout managers can be used in Swing to be more particular about the way that things are laid out in the window. The second example below gives a button that when clicked does nothing.



```
package com.developical.gui;

import javax.swing.*;

public class GuiApp {

    private void display() {
        JFrame frame = new JFrame("Example App");
        frame.setSize(300,200);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new GuiApp().display();
    }
}
```

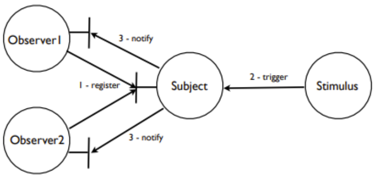
Allows the window to be visible



```
public class GuiApp {

    private void display() {
        JFrame frame = new JFrame("Example App");
        frame.setSize(300,200);

        JPanel panel = new JPanel();
        panel.add(new JTextField(10));
        panel.add(new JButton("Press Me!"));

        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }

    public static void main(String[] args) {
        new GuiApp().display();
    }
}
```

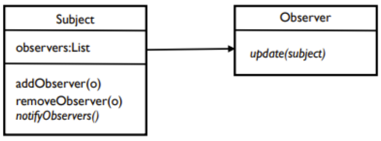
Events: we need a different approach to programming in response to interactive events. We want to give control to the user by executing code in response to actions in the UI, rather than keeping control in the heart of the program.

Observer : pattern

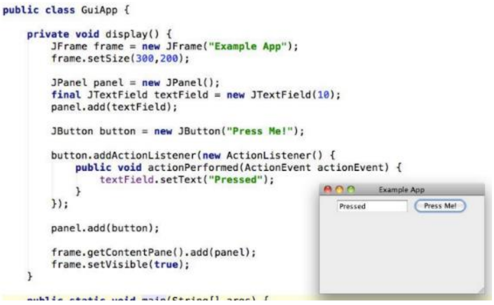


Observer Pattern (aka publish-subscribe): we invert control of the program, so that instead of calling the widget to ask whether it has been pressed, we register an object to be notified when something of interest happens. We can register several observers, all looking at the same subject, and each gets informed of changes. The Observer has two participants: the subject (where we can register observers), and the observer (notified when there is a change of state that is of interest). The subject loops through all its registered observers when there is a change of state and sends an event to each observer (potentially a representation of the state that has changed. The Observer pattern is only commonly used with GUI components.

Observer : Example

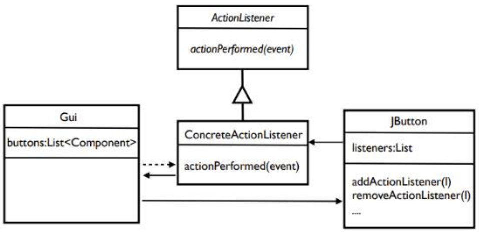


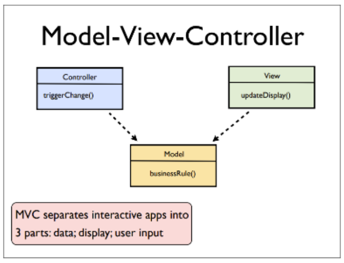
In this example, an Observer pattern has been set up with the JButton as the subject and the ActionListener as the observer. Using an anonymous inner class to implement the ActionListener is idiomatic in Java and Swing, so it is defined inline and the reference to the new ActionListener passed to the button to register it as an observer. When the button is clicked, the text in the text field is updated to show it has been pressed. The diagram shows the relationships between the objects in this example.



```
public class GuiApp {

    private void display() {
        JFrame frame = new JFrame("Example App");
        frame.setSize(300,200);

        JPanel panel = new JPanel();
        final JTextField textField = new JTextField(10);
        panel.add(textField);

        JButton button = new JButton("Press Me!");

        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent actionEvent) {
                textField.setText("Pressed");
            }
        });

        panel.add(button);

        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }
```

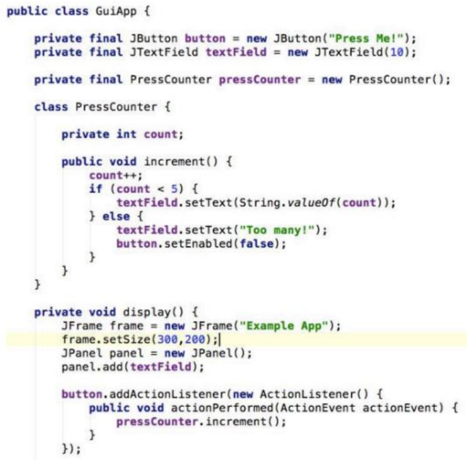
Observer : Swing Button





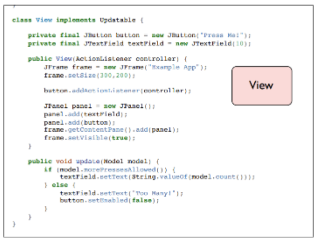
MVC separates interactive apps into 3 parts: data; display; user input

Model-View-Controller (MVC): one of the most common architectures for a GUI application. MVC splits the data model from the view which displays the data. Several views can be built on top of the same data (just as you could use different types of graphs to represent the same data). The controller makes updates to the model by calling methods on its objects in response to external events (e.g., clicks on the UI). The controller may also trigger the view to redraw itself after an event.



```
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() {
        JFrame frame = new JFrame("Example App");
        frame.setSize(300,200);
        JPanel panel = new JPanel();
        panel.add(textField);

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

In this code example, all the elements of MVC are a bit mixed together. We can follow a process similar to below to tidy things up.

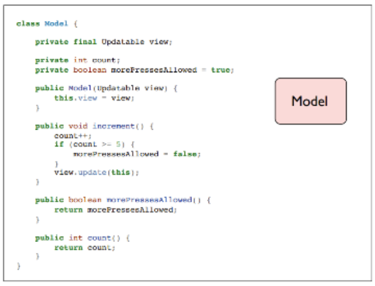


```
class View implements Updatable {

    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(300,200);

        button.addActionListener(controller);

        JPanel panel = new JPanel();
        panel.add(textField);
        panel.add(button);
        frame.getContentPane().add(panel);
        frame.setVisible(true);
    }

    public void update(Model model) {
        if (model.morePressesAllowed()) {
            textField.setText(String.valueOf(model.count()));
        } else {
            textField.setText("Too Many!");
            button.setEnabled(false);
        }
    }
}
```

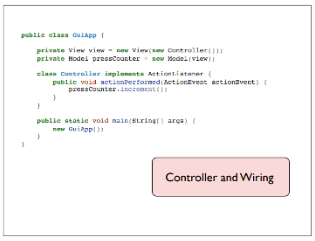
Here, the display logic has been separated out into its own class. The name View is only chosen to more clearly show the MVC pattern, but a more domain-specific name (e.g., Display or GUI) would be more suitable. The View contains the button, and a link to the Controller is set up so it is fired off when the button is pressed. The View also has an update() method, to allow for it to redraw the display as appropriate depending on updates to data in the Model.

The Model class stores the data, and the logical operations that act upon this data. It also contains query methods which allow the View to read the relevant parts of the Model to render the data.

Remember: commands update the model, whereas queries allow access to data values.



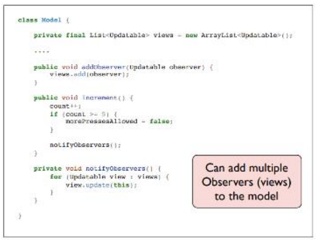
```
class Model {

    private final Updatable view;

    private int count;
    private boolean morePressesAllowed = true;

    public Model(Updatable view) {
        this.view = view;
    }

    public void increment() {
        count++;
        if (count >= 5) {
            morePressesAllowed = false;
        }
        view.update(this);
    }

    public boolean morePressesAllowed() {
        return morePressesAllowed;
    }

    public int count() {
        return count;
    }
}
```



```
public class GuiApp {

    private View view = new View(new Controller());
    private Model pressCounter = new Model(view);

    class Controller implements ActionListener {
        public void actionPerformed(ActionEvent actionEvent) {
            pressCounter.increment();
        }
    }

    public static void main(String[] args) {
        new GuiApp();
    }
}
```

The GuiApp class puts everything together, creating the Model, View, and Controller objects and wiring them together to set up the structure we want. It also contains the main() method that we use as an entry point to the application. Once main() has finished executing, all further action is triggered by events.

In the previous example we could only have one View associated with the Model, but one of the strengths of MVC is that we can create multiple different Views to allow the Model to be displayed in different ways.

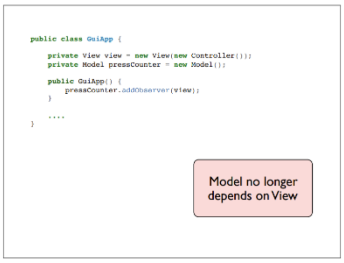


```
class Model {

    private final List<Updatable> views = new ArrayList<Updatable>();

    ....

    public void addObserver(Updatable observer) {
        views.add(observer);
    }

    public void increment() {
        count++;
        if (count >= 5) {
            morePressesAllowed = false;
        }

        notifyObservers();
    }

    private void notifyObservers() {
        for (Updatable view : views) {
            view.update(this);
        }
    }

}
```

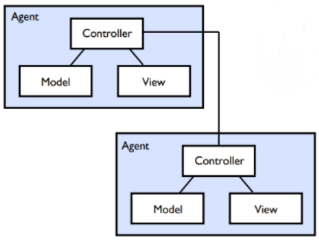
Here we change the way the Model is coded to allow for adding several different Updatables as observers and have each one be notified when there is a change to the Model's state.

Another improvement we can make is to change the way that the objects depend on one another. Previously we couldn't construct the Model without passing it a View (or at least an Updatable) ad a constructor parameter. This makes the Model quite immobile as we can't easily reuse it in a different context.

By changing to this subscription model, where zero or more observers can be added to the Model after construction, we improve the mobility of the code.



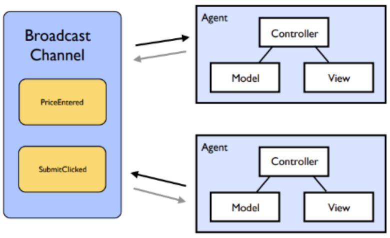
```
public class GuiApp {

    private View view = new View(new Controller());
    private Model pressCounter = new Model();

    public GuiApp() {
        pressCounter.addObserver(view);
    }

    ....
}
```



Presentation-Abstract-Control (PAC): an alternative to MVC, though it is used less often. PAC architectures are well-suited to GUIs where there is a hierarchy to the UI (panels contain sub-panels, each of which has a group of controls or displays for a certain item of data. PAC defines a set of "mini-MVC" agents, which are formed into a tree, with communications between widgets on the page going via connections between the controllers. Communication should only be done up and down the tree, rather than jumping across to other branches.

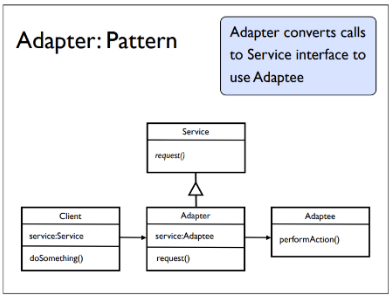
For example, when changing IDE settings, we change one setting and this must travel down the tree to signal to all sub-components that this change has been made. Once the change has been registered at all sub-levels, the "Apply" button at the top level of the hierarchy becomes enabled to allow the user to finalise the changes made.

Event Bus: using PAC means that sometimes we want to send a message up and down the tree to an agent on the other side, requiring a good deal of wiring. An alternative to PAC is to use an Event Bus to allow the agents to communicate. Here, whenever any agent has a change (triggered by the user) it can publish an event to the event bus letting any subscribers know what has happened. Communication then entails an agent publishing an event for another to listen out for. Despite the simpler communication, we have lost the tree structure that we had in PAC, so it can be hard to reason about what will happen in response to an event.



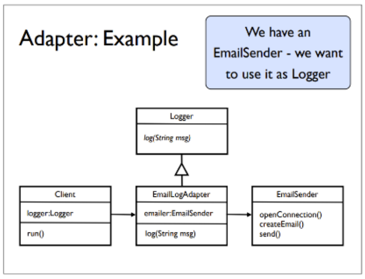
SYSTEM INTEGRATION

When creating new software, we often want to make use of existing components – whether these are part of the existing system we have, or third-party libraries we wish to use. Quite often these won't exactly fulfil the needs we have – perhaps it has the correct behaviour but has the wrong interface.


Adapter: Pattern

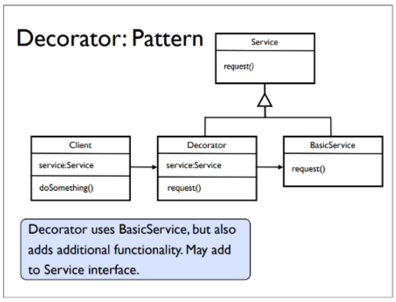
Adapter converts calls to Service interface to use Adaptee

Adapter Pattern: when we have a class, or a service, which can do what we need, but we need it to present a different interface to work with the rest of our system we can make use of the Adapter pattern. We convert the interface of a class into another interface which clients expect. If we cannot change something to fit our needs, we create a wrapper around it.


Adapter: Example

We have an EmailSender - we want to use it as Logger

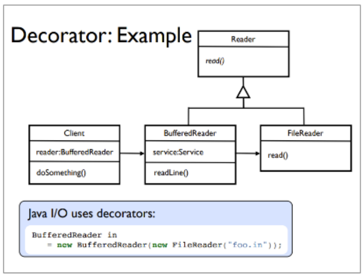
The adapter simply implements a service interface but has no behaviour of its own – it simply exists to translate calls through to the adaptee.


Decorator: Pattern

Decorator uses BasicService, but also adds additional functionality. May add to Service interface.

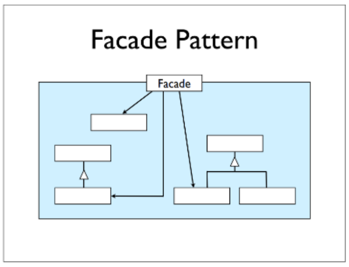
Decorator Pattern: when we have an object we can use, and it has the right interface, but we would like to add additional functionality or responsibility to this object dynamically. We again make use of the technique of wrapping the existing object.

The basic service and the decorator commonly implement the same interface. If this is the case, a decorated or undecorated object can be used interchangeably as far as the client is concerned. If we do wish to dynamically add extra behaviour to the object we are using, the decorator will forward calls to the underlying object, but may perform extra processing before or after those calls.


Decorator: Example

Java I/O uses decorators:

```
BufferedReader in
   = new BufferedReader(new FileReader("foo.in"));
```


Facade Pattern

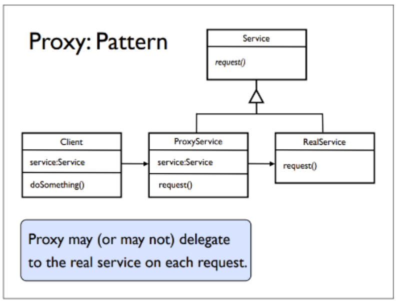
Façade Pattern: sometimes we have a very complex object, or sub-system, which has many possible behaviours, but we only need to use part of it. Covering up some of the complexity that we don't need makes things simpler to understand. This can be achieved using a Façade pattern.

Again, this involves wrapping several underlying objects which provide the service and delegating to them as necessary. The façade normally does not add behaviour, it just coordinates between underlying objects, or hides some of their complexity by not revealing all their possible methods.

Simplicator: a variation on the façade, when communicating between systems rather than individual objects. It has the same motivation as the façade – making interactions with other systems simpler and providing a nicer interface that is easier to work with.

Typically, we may implement a Simplicator as an HTTP service, which connects to another remote service using a more complex protocol.

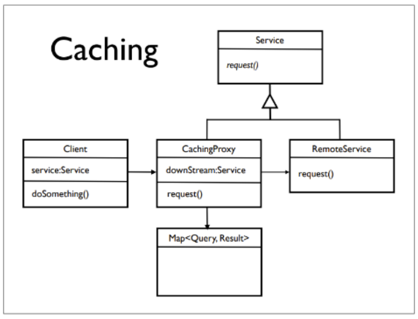
Proxy Pattern: sometimes, when we are using data from a remote service, but it is too slow. We use a Proxy to add to the system something to make the service more responsive (or appear to be so). A Proxy can interpose between the client and server. This may serve several different purposes. Again, this uses the technique of delegation, where the


Proxy: Pattern

Proxy may (or may not) delegate to the real service on each request.

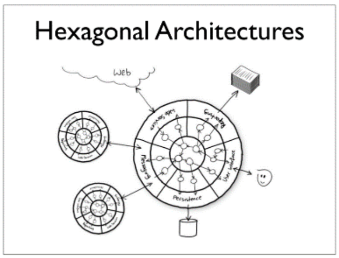
Proxy does not actually implement the service, it delegates all that it cannot do to the server, presenting a more responsive front to the client.

We use the Proxy in place of the real service – we are controlling access to an object (the service) by providing a placeholder or surrogate object. This makes no difference to the client.


Caching

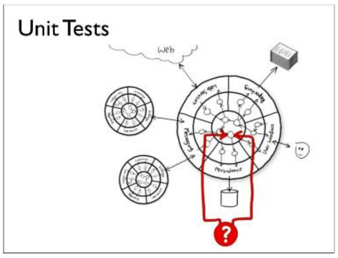
If we are using a service that has high latency, we may be able to cache results to reduce the latency of subsequent calls to the service. We use a proxy that maintains a map of query parameters to results, so for each request we can look it up in the map, and if it is there we return the result without going to the downstream service. If the request is not in the map, once the downstream service is called, the result is stored in the cache before being returned. Data may need to be expired from the cache when it gets stale, or the cache gets too large.

Caching is so commonly used in some environments that we can make use of off-the-shelf products. For example, we often use HTTP caches (e.g., squid or varnish) together with HTTP web sites or services. We could implement a Simplicator as an HTTP service and speed up its performance by inserting an off-the-shelf HTTP cache.

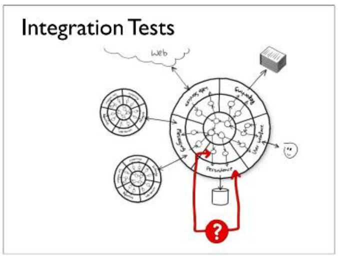

Hexagonal Architectures

Hexagonal Architectures (aka Ports and Adapters): the idea of this architecture is that it separates the core application logic from particular services on which the application depends. These services can only be accessed through a set of adapters, so there is no direct dependency of the core of the system on another system or library from a third party. This isolation makes it easier to swap one implementation of the third-party service/library for another without impacting the core of the application.

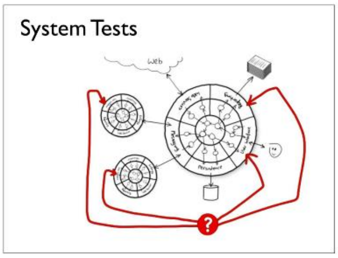
Different sorts of tests can be used to effectively test the system at different levels, each giving confirmation that different aspects of the system work correctly.


Unit Tests

Unit Tests: these test individual elements at the core of the application in isolation, without any dependence on external components. Any external services can be mocked at the adapter level, and the mock plugged in to the port during testing. We typically expect a large suite of unit tests covering many cases, yet which runs quickly (for even thousands of tests), so that they can be run often during development.


Integration Tests

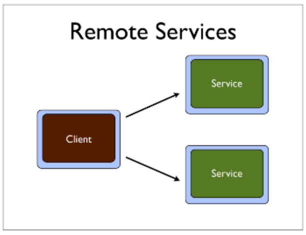
Integration Tests: these test some code we have written against code from a third party, to check the integration works correctly. This is how we might test our adapters, hence these are not used to test the logic of the application, just that the basic connections and translations to external services are working correctly. For example, an integration test for a MySQL database adapter might create some objects, save them into the database, then run some queries to make sure the relevant objects can be retrieved.


System Tests

System Tests: these give us information about whether the whole system is wired together correctly, and that data flows between all the relevant components in the expected way. We may have a relatively small number of system tests, as these are typically slower to run, and just test a couple of the main scenarios end-to-end. These tests are usually run as a final check of the configuration before declaring a potentially releasable version. We would not expect these to check every possible behaviour of the system, as that would typically be a slow and awkward way to test.

DISTRIBUTION AND WEB SERVICES

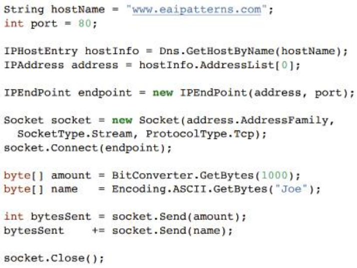
The Internet gives us a way of finding and re-using code made by other developers, and to re-use components that operate over the network. We call these services. As well as combining components and services to build up applications, we may also wish to split up our applications into parts, whether this be for performance reasons (splitting work over multiple machines) or for easier management and faster development (splitting the work over smaller teams of developers).


Remote Services

When splitting an application, it is common to have some components that provide services, and others that are clients of these services. This may extend to a complex graph, where clients of one service provide their own services to other clients higher up in the chain. Clients and servers may run on the same machine, or on different ones. Ween we communicate with other machines; we have additional complexities to consider caused by networking.

Microservice Architectures: there is a general trend in the industry to move towards microservice architectures, where each application is built up from a relatively large number of small services that cooperate to provide the full system.

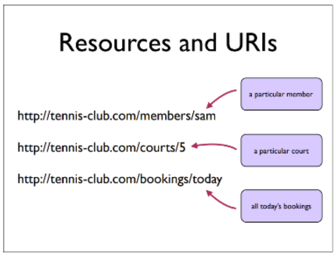
At a low level we can create a connection to a remote machine using a socket and send bytes of data over that socket to the remote machine for it to interpret. This can be problematic as it depends on us knowing (or assuming) a lot of detail about the implementation of the receiving process, for example whether the receiver is the same number of bits (32- or 64-bits) as the sender, or that the receiver knows how to correctly interpret a string of bytes and turn it back into a functioning object.



```
String hostName = "www.eaipatterns.com";
int port = 80;

IPHostEntry hostInfo = Dns.GetHostByName(hostName);
IPAddress address = hostInfo.AddressList[0];

IPEndPoint endpoint = new IPEndPoint(address, port);

Socket socket = new Socket(address.AddressFamily,
  SocketType.Stream, ProtocolType.Tcp);
socket.Connect(endpoint);

byte[] amount = BitConverter.GetBytes(1000);
byte[] name   = Encoding.ASCII.GetBytes("Joe");

int bytesSent = socket.Send(amount);
bytesSent    += socket.Send(name);

socket.Close();
```

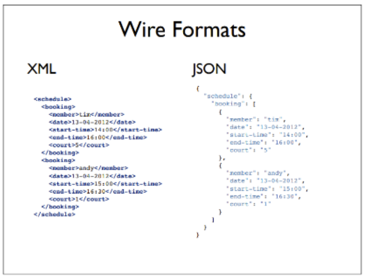
Rather than rely on this low-level networking, it has become common to use the mechanisms proovided by the web as a means of communicating between computers and applications.

HTTP (Hypertext Transfer Protocol): the basic transport protocol used when building web services, which allows us to transfer documents between computers over a network. It has two other useful features – methods (give meaning to the type of request being made, e.g., GET and POST), and status codes (allow the sender to know whether or not their request was successful, e.g., 200 and 403).


Resources and URIs
http://tennis-club.com/members/sam
http://tennis-club.com/courts/5
http://tennis-club.com/bookings/today
a particular member
a particular court
all today's bookings

REST (Representational State Transfer): the architectural style for building services based on the principles of the web. REST services are built around the idea of resources and representations . Resources are things in the world (either physical or conceptual), and are identified by URIs (Uniform Reource Indicators – similar to URLs). We transfer a representation of a rescourse between computers over the network in order to communicate between different services in our system.

There can be different representations of the same resource, expressed in different data formats. Human-readable formats include XML and JSON (shown to the left), and it is sometimes possible to request a particular representation. Some services use binary formats like Protocol Buffers, Thrift, or Avro to pack in data more tightly. These formats are all platform independent so can be used as long as you have a suitable parser at both ends.

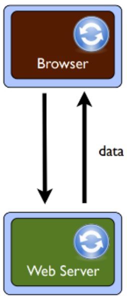

Wire Formats

XML

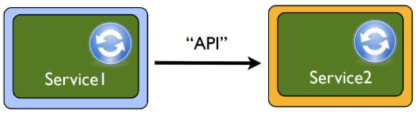
```
<schedule>
  <booking>
    <member>tim</member>
    <date>13-04-2012</date>
    <start-time>14:00</start-time>
    <end-time>16:00</end-time>
    <court>5</court>
  </booking>
  <booking>
    <member>andy</member>
    <date>13-04-2012</date>
    <start-time>15:00</start-time>
    <end-time>16:30</end-time>
    <court>1</court>
  </booking>
</schedule>
```

JSON

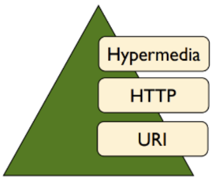
```
{
  "schedule": {
    "booking": [
      {
        "member": "tim",
        "date": "13-04-2012",
        "start-time": "14:00",
        "end-time": "16:00",
        "court": "5"
      },
      {
        "member": "andy",
        "date": "13-04-2012",
        "start-time": "15:00",
        "end-time": "16:30",
        "court": "1"
      }
    ]
  }
}
```



Now that we can build a service that produces JSON, we can consume this from a client application, e.g., a rich JavaScript application running in a browser. This could be fetched asynchronously in the background to update the display the user sees. Alternatively, it may be another server-side application that is calling our web service.



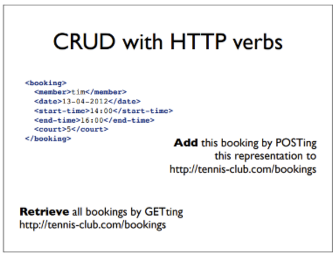
We can create HTTP clients in most programming languages, for example in Java with Apache commons HttpClient.



Richardson Maturity Model: used to describe and categorise webservices based on the degree to which they take advantage of each of URIs for identifying resources, HTTP, and the various methods that it supports, and hypermedia in terms of linked data.

Level 0: services which use HTTP as a means of transporting data, without taking advantage of URIs to identify resources, HTTP methods to describe actions, or hypermedia. These normally use a single URI to identify the service "endpoint" to which requests are posted. Each request consists of a document or a set of paramters that describes the request. An example of this is using a SOAP (Simple Object Access Protocol) envelope to wrap an XML document describing a request. WDSL attempts to describe the expected protocol by a service and the format of request/reponse documents.

Level 1: services which make use of more URIs to represent different types of resources in the system, but typically do not take advantage of available HTTP methods. They do not respect the correct semantics for HTTP methods (.e.g, they can often use GET requests to cause side-effects on the system state). Level 1 service's URLs are verbs and not nouns – that is, they do not identify resources, they correspond only to actions. These URLs may all respond to GET requests with additional parameters being passed after the ?.



CRUD with HTTP verbs

```
<booking>
  <member>tim</member>
  <date>13-04-2012</date>
  <start-time>14:00</start-time>
  <end-time>16:00</end-time>
  <court>5</court>
</booking>
```

Add this booking by POSTing this representation to http://tennis-club.com/bookings

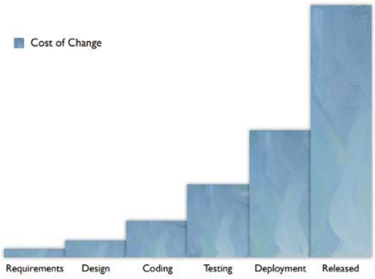
Retrieve all bookings by GETting http://tennis-club.com/bookings

Level 2: services which use URIs to represent different types of resources, and also respond to different HTTP methods (typically GET, POST, PUT, and DELETE) in order to update the state of these resources. They send appropriate HTTP status codes with their responses, allowing the client to track the effects of calls they have made. Note CRUD = Create, Read, Update, and Delete.

Level 3: the highest level in the Richardson Model. Characterises fully RESTful services, building on Level 2 services. The key point about Level 3 services is that the representations that they use contain hyperlinks to other resources that the client can follow.

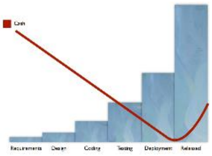
Level 2 vs Level 3: with Level 2 services, clients often follow URI templates to construct the URI for a particular resource, which reveals more about the implementation than we would like. Level 3 services might allow users to do a search and get back a document containing links to the user records (for example) which the client can follow as opposed to assuming the structure of those links.

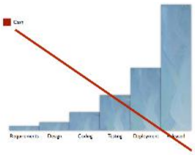
CONTINUOUS DELIVERY AND AGILE METHODS



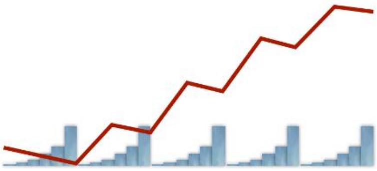
This graph shows the Cost of Change curve. It shows how a change of requirements made early is relatively cheap to implement, however as we move through the project, the cost of making changes to the code grows to be very expensive. Spending lots of time at the start of a project ensuring the analysis and requirements are all correct rarely works out well.

A problem when releasing at the end of a project, once everything is finished, is that the project returns no value until the product releases. This is risky – what if the software does not sell many copies? Or the system itself is broken upon release?





What if you're a sad sack of shit who can't code to save your life and no one buys/uses/downloads your product?



Companies these days very rarely use the waterfall model of development, instead favouring agile, iterative methods, and to revise and refine the design of their software constantly as new features are added. Working in short iterations allows for requirements to change/be prioritised differently over time depending on the needs of the customers. These cycles range from a few days to a few weeks each.

Agile methods favour an iterative approach – rather than proceeding in phases, we iteratively design, build, and release small sets of features, aiming to deliver value from the first release (which should be early in the project). By compressing the development waterfall into a succession of small cycles, performed repeatedly, the time between someone having an idea and it being implemented is reduced.

The first release happening sooner (of a high-value feature, preferably) allows for the project to begin generating revenue sooner. If the project does not look to be profitable after the first couple of releases, it can be scrapped (early!), and the iterative cycle allows for the team to learn quicker what the users want in the software.

The Agile Manifesto: gives some overriding principles which can lead to effective software development practice, focussing on improving the way people interact and communicate to build the right software (rather than necessarily building processes and tools). The Manifesto favours delivering working software over writing complex design documents, and that having a plan for a project is good, but we should be happy to change the plan if/when needed.

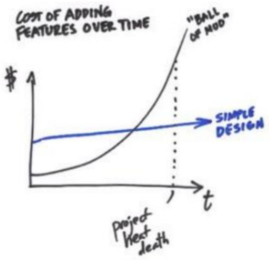
Some other examples of development methods include:

Extreme Programming (XP): one of the original agile methods which includes project management techniques as well as technical practices for delivering reliable software quickly.

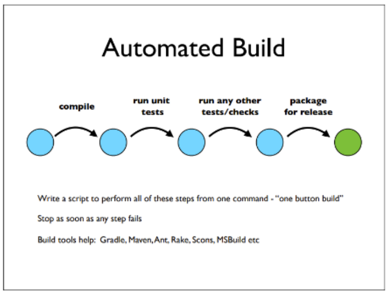
Scrum: concentrates more on project management methods as opposed to talking specifically about building software.

Kanban: a more recent method which discards the timeboxed iterations from XP and Scrum, aiming for a continuous flow of work.

If we want to continually change our system, we must pay attention to the design concerns looked at in this course. Paying attention to improving design over time allows us to maintain the software effectively and reduces the cost of adding new features. Successful agile development relies on following a good set of design techniques and principles covered in this course.



To squeeze the development cycle and still deliver reliable software, we define the smallest possible amount of functionality possible for each idea that it makes sense to deliver (something which can be implemented with a few days' work). We then follow TDD to ensure the quality of the work and prevent regressions, and at the end of the iteration should hopefully have a new version of the product ready to release.



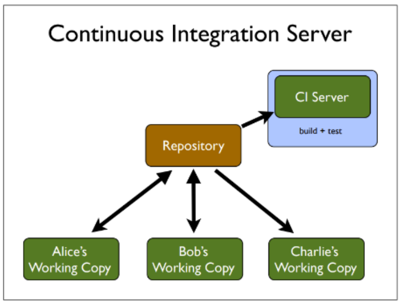
Write a script to perform all of these steps from one command - "one button build"

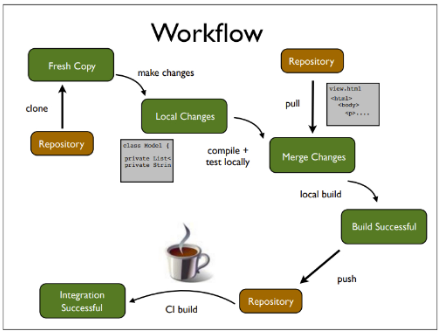
Stop as soon as any step fails

Build tools help: Gradle, Maven, Ant, Rake, Scons, MSBuild etc

Continuous Integration: each developer should merge all their work into master at least once per day (ideally more frequently than this) to avoid merging long-lived branches at the end of development and facing painful integration problems. We work in small, non-breaking changes, keeping the master branch in a state where it could be deployed at any time. Typically, an automated build is run (which includes all our tests and checks) before an after each push, to keep the master branch in a healthy state.

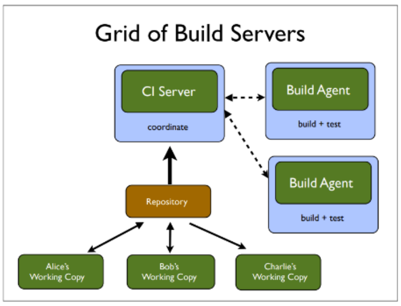
Continuous Integration (CI) Server: on larger projects with many frequent commits, it can be difficult to run a full build locally in every commit. A secondary check (and perhaps a larger test suite) can be run by a CI server, which can also gather data and statistics on changes made, test failures, fixes, etc. The CI server can be set up to build on a regular schedule, or to run when it detects a change in the repository (i.e., when a new commit is made).



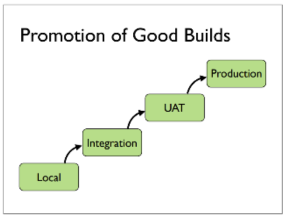


When adding a CI server, our workflow is extended so that the CI build runs automatically after changes are committed. Once we have confirmed that the CI build was successful, we can safely move on to the next feature. Teams often use a visual signal, such as a display screen, to show the status of their builds (and perhaps others which they depend on). Being able to clearly see when a certain build fails is useful as the team can then turn their attention to fixing that before continuing work on new features.

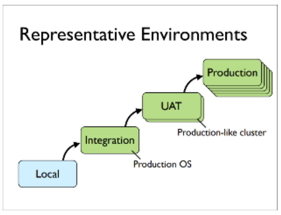
For teams with a lot of CI build to run (perhaps testing on different platforms), a grid of CI servers or agents can be useful. CI servers such as TeamCity or Jenkins support this without too much work. Parallelising the builds and farming them out to different agents speeds up the rate of obtaining feedback.

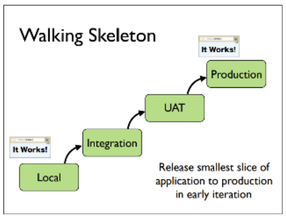


Release is often one of the points of the project that causes the most stress, and where problems come up and fixes have to be made after pushing software into production.



Instead of putting code straight into production, we want to test it first in various pre-production environments, and promote builds through this chain when we are confident that they are of good quality. Often developers' local environments differ substantially from production environments. They may differ in architecture, OS, network infrastructure, etc. We aim to test in a production-like environment as early as possible.




Walking Skeleton
It Works!
Production
UAT
It Works!
Integration
Local
Release smallest slice of application to production in early iteration

Walking Skeleton: deployed in the first iteration of project work to help with ironing out difficulties incurred when releasing software to production. If we leave solving problems (which cause delays when the software is due to be released) right to the end of the project, this has a big effect on whether or not the project can be delivered on time.

The automation used with CI can be extended so that if we pass all the tests and checks, we automatically deploy the code – likely first into a test environment, and if that works well we can then promote the code into production. Depending on the size of the team, the product, and the risk associated with bugs (e.g., for a web app, how big a problem would it be if the app went down?) we may be more careful or aggressive about pushing changes to production, and have approproate numbers of steps and checks in our pipeline for these cases.

However, in many organisations, getting to the point where fully automated deployments can be done is hard, and not normally a technical problem.