SED

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.

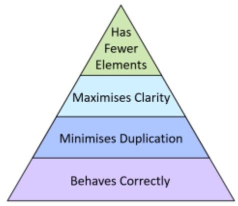
Waterfall development:
Requirement gathering → Analysis → Design → Coding → Testing → Deployment
Not used these days, little scope for going back and reworking things

TDD (Test Driven Development):
Write a **failing test** → Write **enough code to pass** the test → **Refactor** → **Repeat**
In other words:
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)

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

BDD (Behaviour Driven Development):
Write a few behavioural properties of the object we want to create (an **informal specification**) → Translate **a requirement into unit tests** → Write **code to pass** → **Refactor** → Repeat for **next requirement** in informal specification

Four Elements of Simple Design



Refactoring methods:
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.
Removing Duplication Between Classes: first refactor similar things to be the same, then refactor away these commonalities (perhaps into another object/class).
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. (Extract an interface, implement versions with behaviours depending on the type).

JUnit Testing/Mocking:

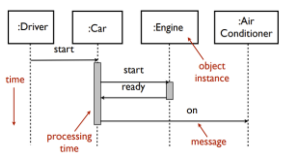
```
public class FibonacciSequenceTest {

    @Test public void
    definesTheFirstTwoTermsToBeOne() {
        ...
    }
```

CoreMatchers

```
@BeforeEach
void setup() {
    calculator = new Calculator();
}
assertTrue(x); assertEquals(x, y);
assertFalse(x);
assertThat(x, is(y));
```

Mock objects:
Command: tell another object to do something for us (delegate responsibility), do not get return value, often change the state of the invoked object or other part of program. Tell don't ask!
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. Tell don't ask!
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.
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.
Collaborate Through Roles: Java uses Interfaces to represent roles.

UML Sequence Diagram




```
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);
    }
}
```

@Rule
Make the things used in the test

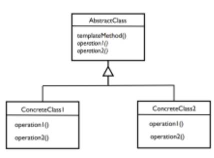
Test Setup
Expectation
Trigger

will(returnValue(VAL_TYPE)); (after exactly)

Designing for Flexibility:
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 avoids fragility (limits blast radius), information hiding helps abstraction
public/protected things are part of the class' API – cannot be changed after deployment
Train wreck: getX().getY().getZ().doSomething()
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.

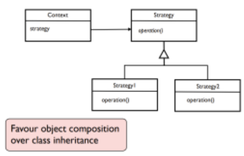
Open-Closed Principle: a class's behaviour should be extensible without modification.
Coupling: close coupling (e.g., inheritance like in template method) causes immobility (if want to use subclass, have to bring superclass too)

Template Method : pattern



Extract commonality to a superclass (abstract class) then inherits from that class and implement abstract method(s)).

Strategy : pattern



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

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: balance of an object's independence and responsibility. Objects at the core of the system (highly depended on) should be stable, vice versa for unstable (frequently changing).
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). Software such as NDepend.
McCabe Complexity: gives lower bound for number of tests required for a unit, by counting the numbers of nodes and edges in the CFG for program + counting possible different executions.
WILT: Whitespace Integrated over Lines of Text for a given piece of code, integrating over indented area gives measure of complexity. Strong correlation between WILT and McCabe complexity. (WILT much cheaper, McCabe very expensive).
ABC Metrics: count the number of Assignments, Branches, and Conditions.
Lifelines: plot the complexity of code over time.
Turbulence: plotting the number of commits made to each file against their complexity → 4 quadrants, Simple/Complex and Seldom change/Often change.
Temporal Coupling: When things change at the same time. Track files changed in same commit.

Factory Methods

Java

```
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;
```

private constructor forces callers to use factory methods

Abstract Factory Pattern

```
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 {
```

Builder Pattern: REMEMBER THE PIZZA!
BananaBuilder has static method aBanana() which makes a new BananaBuilder
.withRipeness() etc take the attribute's value, set the BananaBuilder's field to that value then return a BananaBuilder
.build() then makes the Banana with all the fields of the BananaBuilder (with nulls/defaults where needed already set as default in the Builder)

```
import static BananaBuilder.*;

public class FruitBasket {

  private Collection<Fruit> basket = new ArrayList<Fruit>();

  public FruitBasket() {

    Banana banana = aBanana().withRipeness(2.0).withCurve(0.9).build();
```

Singleton

if you really need to ensure that everyone is using the same object

```
public class BankAccountStore {

    private static BankAccountStore instance = new BankAccountStore();

    private Collection<BankAccount> accounts;

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

    public static BankAccountStore getInstance() {
        return instance;
    }

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

static initialisation runs when class is loaded
private constructor enforces that no-one else can call new BankAcountStore()
clients call getInstance() method to gain access to the global instance

Singleton: USE SPARINGLY! Only need to use when you absolutely need to, sometimes you happen to only have one thing and that is FINE. Can use singleton when the object takes a long time to instantiate and won't change. Singleton introduces global variables (more coupling). Passing in the singleton and using an interface to define the singleton's behaviour reduces the direct dependency on the singleton (reduced coupling).

Legacy Software:
Seams: to 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. We can use **object seams** to test effects of the code on databases/emails sent.
Enabling Point: the point where you decide to use one behaviour or the other (i.e., when we pass in our test implementation as opposed to the real implementation of the dependency). May not be possible if the code being tested uses the new operation to create its dependency or refers to a singleton instance.

Concurrency:
Most common way: class **extending** Thread class, override run() then call start() on it after constructing. Introduces close coupling due to inheritance.
Runnables: compose objects using Strategy pattern, create a runnable (**implements** Runnable) then implement the run() method, then can either call run() (async) or make a new Thread, passing in the runnable, and call start() (sync).

```
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);
```

```
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;
  }
}
```

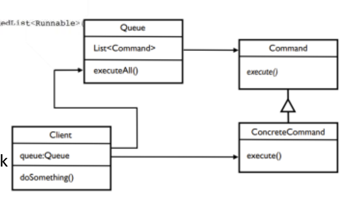
like runnables but allow values to be returned from the call() method and allow exceptions to be thrown if necessary

Callables

```
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();
        }
    }
}
```

The Command pattern can be used with both Runnables and Callables as they are both examples of a device where a piece of behaviour can be wrapped up, passed around + executed in different contexts.

Command: Pattern



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

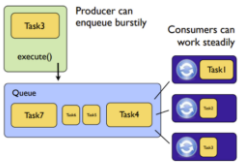
Concurrency Continued:
Executors: manage a pool of threads to execute enqueued tasks concurrently. Producers call execute() on tasks which enqueue the task waiting to be send to Consumers (threads in the pool) to be executed.

```java
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");
        }
```

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.

Queues are Load Balancers




```java
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");
}
```

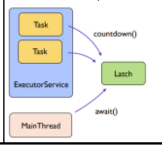
ExecutorService: provide more functionality than a basic Executor. Allow us to wait for all tasks to complete (then do something else like inform the user that the tasks are done). In the code, after submitting all tasks, we tell the executor to shutdown, and then wait for termination. Once this call returns, we know the executor's queue is empty (or the timeout has expired).

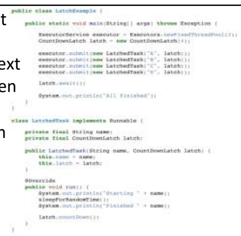
Futures: when submitting a task to an executor using a Callable, it returns a Future. Futures can be used to track the execution of our task. There are many different methods on Futures such as get(), get(timeout), cancel(mayInterruptIfRunning), isDone(), and isCancelled().

```java
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);
}
```

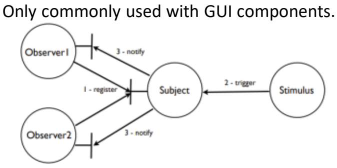
Block until the result is available

Latches: useful 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.

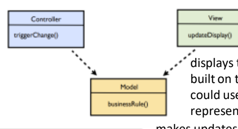




Interactive applications:
Observer pattern (publish-subscribe): invert control of the program - instead of calling the widget to ask whether it has been pressed, we register (one or many) object(s) to be notified when something of interest happens. Has two participants: the subject (where we can register observers), and the observer (notified when there is a change of state of interest). Only commonly used with GUI components.



Model-View-Controller



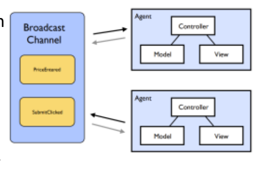
MVC separates interactive apps into 3 parts: data; display; user input

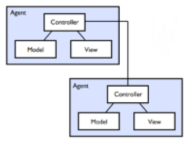
Model-View-Controller (MVC): one of the most common architectures for a GUI app. Splits the data model from the view which displays the data. Several views can be built on top of the same data (just as you could use different types of graphs to represent the same data). The controller makes updates to 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.

```java
public class GuiApp {
private View view = new View(new Controller());
private Model pressCounter = new Model();
public GuiApp() {
pressCounter.addObserver(view);
}
....
}
```

View = display logic,
Model = data+logical ops on data
Controller = actionPerformed

Presentation-Abstract-Control (PAC): alternative to MVC suited for GUIs with a hierarchy in the UI (e.g., 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.



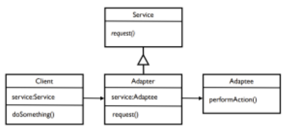


Event Bus: using PAC means sometimes we want to send a message up and down the tree to an agent on the other side, requiring a good deal of wiring. An alternative to PAC is to use an Event Bus to allow the agents to communicate. When an agent has a change (triggered by the user) it can publish an event to the event bus letting subscribers know what happened. Loses the PAC tree structure so harder to reason about what happens after an event.

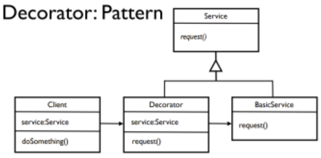
Adapter Pattern: used when we have a class or a service which can do what we need, but we need it to present a **different interface** to work with the rest of our system. We convert the interface of a class into another interface which clients expect. If we cannot change something to fit our needs,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

Adapter: Pattern

Adapter converts calls to Service interface to use Adaptee



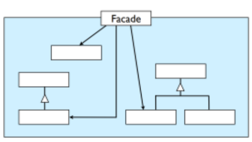
Decorator: Pattern



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

System Integration:
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 wrap 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.

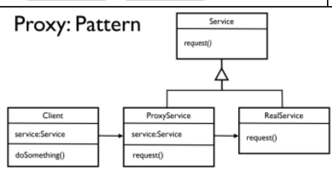
Facade Pattern



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. The façade pattern helps to cover up some of the complexity that we don't need to make things simpler to understand. 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 is still about 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.

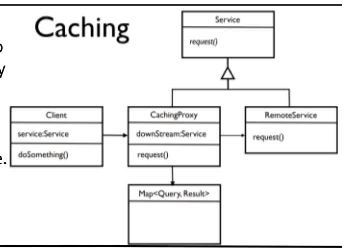
Proxy: Pattern



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

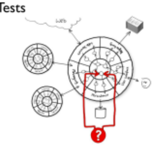
Proxy Pattern: can interpose between the client and server and may serve several different purposes using the technique of **delegation**, where the Proxy does not actually implement the service, delegating all that it cannot do to the server. If we are using a service that has high latency, we may be able to **cache** results to reduce the latency of subsequent calls to the service using 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: 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. Could implement a Simplicator as an HTTP service and speed up its performance by inserting an off-the-shelf HTTP cache.

Caching



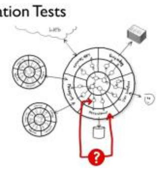
Hexagonal Architectures (aka Ports and Adapters): separates the core application logic from particular services on which the application depends. These services can only be accessed through a set of adapters, so there is no direct dependency of the core of the system on another system or library from a 3rd party. Isolation makes it easier to swap one implementation of the 3rd party service/library for another without impacting the core of the application. We use unit/integration/system tests to test the different aspects of the system.

Unit Tests



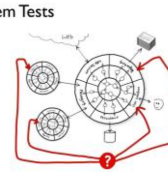
Unit Tests: test individual elements at the **core of the application** in isolation, without any dependence on external components. Any external services can be mocked at the adapter level, and the mock plugged in to the port during testing. Large suite of unit tests, cover wide range of cases, run quickly (even for 1000s of tests), run often during development.

Integration Tests



Integration Tests: testing **adapters** can be done by testing code we have written against code from a 3rd party, to check the integration works correctly. Do not test the logic of the app, only the basic connections and translations to external services. E.g., 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: give 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. 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:
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
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 resourse between computers over the network to communicate between different services in our system.
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: use HTTP as a means to transport data
Level 1: use more URIs to identify types of resources in the system
Level 2: respond to different HTTP methods + return status codes
Level 3: fully RESTful services. URIs cannot be manipulated by the user like in level 2.

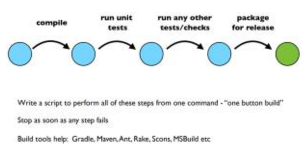
Continuous Delivery and Agile Methods:

Cost of Change

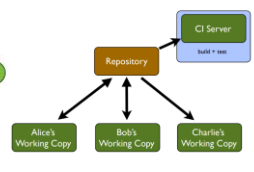
Agile methods: favour iteratively designing, building, and releasing small sets of features, aiming to deliver value from the first release (which should be early in the project). By compressing the development waterfall the time between someone having an idea and it being implemented is reduced.

Walking Skeleton: deployed in the first iteration of project work to help with ironing out difficulties incurred when releasing software to production which should not be left to the end.

Automated Build



Continuous Integration Server



Scrum: concentrates more on project management methods as opposed to talking specifically about building software.
Extreme Programming (XP): one of the original agile methods which includes project management techniques as well as technical practices for delivering reliable software quickly.
Kanban: a more recent method which discards the timeboxed iterations from XP and Scrum, aiming for a continuous flow of work.