# 1 Test Driven Development

```
@Test public void undefinedForNegativeIntergers() {
    try {
        sequence.term(-1);
        fail("should have thrown exception");
    } catch (IllegalArgumentException e) {
        assertThat(e.getMessage(), containsString("Undefined for negatives!"));
    }
}

public class SystemClock implements Clock {
    public LocalTime now() {return LocalTime.now()}}
private class ControllableClock implements Clock {
    LocalTime now = LocalTime.now();
    public LocalTime now() {return now;}
    public void windForward(int i, ChronoUnit units) {now = now.plus(i, units)}
}
```

# 2 Mock Objects

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

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

  @Test public void delegatesPuddingsToPastryChef() {
    context.checking(new Expectations() {{
      exactly(1).of(pastryChef).order("Pudding"); // ignoring() allowing() oneOf()
      will(onConsecutiveCalls(returnValue("Ordered Pudding"), returnValue("Ordered Cake")));
      never(pastryChef).order(with(any(byte[].class)));
    }});

    headChef.order("Chicken", "Pudding");
  }
}
```

# 3 Designing for Flexibility

- **Law of Demeter**: Each unit should only talk to its immediate friends - limiting blast radius
- **Tell, Don't Ask Style**: Command: Delegate work w/o expecting feedback, changes state of invoked object. Query: Request info w/o side effects. → Localises logic and info, simplifies testing. Otherwise fragile + tightly coupled.
- **Bad Design**: Rigidity (hard to understand/easily change - encapsulation), Fragility (other parts break if one part changed), Immobility (hard to reuse code elements in other applications)

# 4 Re-use and Extensibility

- **4 Elements of Simple Design**: Behaves Correctly, Minimises Duplication, Maximises Clarity, Has Fewer Elements
- **Open-Closed Principle**: You should be able to extend a class's behaviour, without modifying it
- **Afferent Coupling**: how many other classes use this class (responsibility)
- **Efferent Coupling**: how many different classes used by this class (independence)

## 4.1 Template Method Pattern

Tight coupling since subclasses cannot be used independently or in another system without the superclass

```
public abstract class NumberSequence {
  public int term(int i) {
    if (i<0) throw new IllegalArgumentException("Negative Undefined");
    return positiveTerm(i);
```

```
  }

  protected abstract int positiveTerm(int i);
}

public class TriangleNumberSequence extends NumberSequence {
  @Override protected int positiveTerm(int i) { return (i+1)*(i+2)/2 }
}
```

### 4.2 Strategy Pattern

```
public class NumberSequence {
  private final TermGenerator termGenerator;
  public NumberSequence(TermGenerator termGenerator) {this.termGenerator = termGenerator;}

  public int term(int i) {
    if (i<0) throw new IllegalArgumentException("Negative Undefined");
    return termGenerator.positiveTerm(i);
  }
}

public class TriangleNumberSequence implements TermGenerator {
  @Override public int positiveTerm(int i) { return (i+1)*(i+2)/2 }
}
```

## 5 Creation and Dependency

### 5.1 Factory Method

```
class VirtualMachine     {
  private final int memory;

  public static VirtualMachine highMemory() {return new VirtualMemory(100);}
  private VirtualMachine(int memory) {this.memory = memory;} // forces factory methods
}
```

### 5.2 Factory Object

```
class LogoFactory implements Supplier<Logo> {
  static Logo get() {
    if (config.country().equals(Country.UK)) return new FlagLogo("Union Jack");
    return new DefaultLogo();
  }
}
```

### 5.3 Builder

```
public class BookSearchQueryBuilder {
  private String firstName = null;
  private BookSearchQueryBuilder() {}

  public static BookSearchQueryBuilder books() {return new BookSearchQueryBuilder();}
  public BookSearchQueryBuilder withFirstName(String firstName) {
    this.firstName = firstName; return this;
  }
  public build() {return new BookSearchQuery(firstname);}
}
```

### 5.4 Singleton

Tight coupling. Harder to test individual components. & harder to swap out components.

```
public class LibraryCatalogue {
  private static LibraryCatalogue instance = new LibraryCatalogue();
  private LibraryCatalogue() {}
  public static LibraryCatalogue getInstance() {return instance;}
}
```

# 6 Concurrency

```
public class LatchedTimedTask implements Callable<Long> {
  private final CountDownLatch latch; private final Runnable task;
  public LatchedTimedTask(CountDownLatch latch, Runnable task) {}
  @Overide Long public call() {
    long start = System.currentTimeMillis();
    task.run(); latch.countdown();
    return System.currentTimeMillis() - start;
  }

  public static void main(String[] args) throws Exception {
    ExecutorService executorService = Executors.newFixedThreadPool(n);
    CountDownLatch latch = new CountDownLatch(n);
    .. Future<Long> future = executorService.submit(latch, () -> function(i));
    executorService.shutdown();
    try {executorService.awaitTermination(120, TimeUnit.SECONDS) or latch.await(2, TimeUnit.MINUTES)}
    catch (InterruptedException e) {throw new CustomException(e);}
    .. Long time = future.get()
  }
}
```

# 7 Interactive Applications

```
public class Calculator implements Updatable {
  private final JTextField output = new JTextField(10);
  // view
  private void display() {
    JFrame frame = new JFrame("Calculator");
    ArithmeticEngine calc = new ArithmeticEngine();
    calc.addObserver(this);
    frame.setSize(350,300);
    JPanel panel = new JPanel();

    panel.add(output);
    addNumberButtons(panel, calc); addOperatorButtons(panel, calc);

    frame.getContentPane().add(panel);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

  @Override public void updateWith(int value) {
    output.setText(String.valueOf(value));
  }

  // controller
  private addNumberButtons(JPanel panel, ArithmeticEngine calc) {
    IntStream.range(1,5).forEach(n->{
      JButton button = new JButton(String.valueOf(n));
      button.addActionListener(e->calc.input(n));
      panel.add(button);
    });
  }

  private addOperatorButtons(JPanel panel, ArithmeticEngine calc) {
    EnumSet.allOf(Operator.class).forEach(op->{
      JButton button = new JButton(op.label());
      button.addActionListener(e->calc.apply(op));
      panel.add(button);
    });
  }

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

```
enum Operator {
  PLUS("+") {@Override public Integer apply(Integer x, Integer y) {return y + x;}},
  MINUS("-") {@Override public Integer apply(Integer x, Integer y) {return y - x;}};

  private String label;
  Operator(String label) {this.label = label;}
  public String label() {return label;}
  public abstract Integer apply(Integer x, Integer y);
}

public class ArithmeticEnginer { // model
  private final List<Updatable> = new ArrayList();
  private final Stack<Integer> = new Stack();

  public void input(int value) {stack.push(value); notifyObservers();}
  public void apply(Operator op) {
    stack.push(op.apply(stack.pop(), stack.pop())); notifyObservers();
  }

  public void addObservers(Updatable observer) {observers.add(observer);}

  private void notifyObservers() { // model should update view directly
    for (Updatable observer : observers) {observer.updateWith(stack.peep());}
  }
}
```

Alternative is Presentation-Abstraction-Control made of mini-MVC agents. The agents are formed into a tree and communication is by controllers up and down the tree. Another alternative is Event Bus.

# 8 System Integration

Hexagonal Architectures

## 8.1 Adapter

Convert the interface of a class into another interface clients expect.

```
public class WeatherDotComTemperatureService implements TemperatureService {
  Forecaster forecaster = new Forecaster();
  @Override // throws IllegalArgumentException
  public int temperatureFor(String place, DayOfWeek day) {
    return forecaster.forecastFor(com.weather.Region.valueOf(place.toUppercase()),
      com.weather.Day.valueOf(day.name().toUpperCase())).temperature();
  }
}
```

## 8.2 Decorator

Add additional functionality or responsibility to an object dynamically.

## 8.3 Proxy

Control access to an object by providing a placeholder or surrogate object.

```
.. public CachingService(Service downstream, int capacity) {
  this.downstream = downstream; this.cache = new LinkedHashMap<Pair<A,B>,C>() {
    // Pair<A,B> must override equals() and hashcode()
    @Override protected boolean removeEldestEntry(Map.Entry eldest) {return size() > capacity;}
  };
}
```