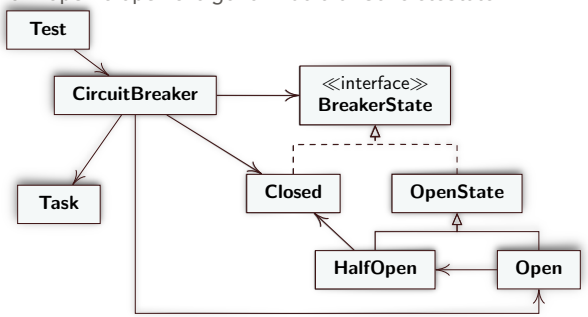
Case Study Circuit Breaker

- Si abbia un servizio (la classe `Task`) che durante l'esecuzione potrebbe essere affetto da errori o da rallentamenti
- Il design pattern CircuitBreaker monitora richieste e risposte e passa dallo stato di chiuso allo stato di aperto, e quindi allo stato di semiaperto, in base ad errori e rallentamenti che si presentano a runtime dal monitoraggio del servizio `Task`
- Il design pattern CircuitBreaker è implementato sulla struttura del design pattern State, quindi la classe `CircuitBreaker` svolge il ruolo di *Context*, inoltre le classi `Closed`, `HalfOpen` e `Open` svolgono il ruolo di *ConcreteState*



Dettagli Del Servizio

- Il servizio da eseguire è implementato nel metodo `nextStep()` di `Task`. Esso può lanciare un'eccezione, o durare più di quanto ci si aspetta (massimo 300 ms), e in questi casi qualcosa non va, oppure ritornare un valore in un tempo minore di 300 ms
- L'implementazione di esempio fa sì che un terzo delle esecuzioni lanci un'eccezione, un terzo duri da 200 a 700 ms, e un terzo ritorni subito


```
public class Task {
   private int step = 0;
   private static final String[] steps = {"stpZero","stpOne","stpTwo","stpThree","stpFour"};

   private int nextStepOk() {
       System.out.print("In next step ... ");
       System.out.print("From " + steps[step] + " to ");
       step++;
       if (5 == step) step = 0;
       System.out.println(steps[step]+" ...");
       return step;
   }

   private static void randomDelay() {
       int delay = 200 + new Random().nextInt(500);
       try {
           Thread.sleep(delay);
       } catch (InterruptedException e) {
          throw new RuntimeException(e);
       }
   }

   /**
    * Statistically, one third of the times the next step cannot be reached, an Exception
    * occurs. When there is no Exception, there is a variable delay.
    * @return a value between 0 and 4 or throws Exception
    * @throws Exception
    **/
   public int nextStep() throws Exception {
       System.out.print("Try to go to next step ... ");
       int randomVal = new Random().nextInt(3);
       if (0 == randomVal) throw new Exception();
       else if (1 == randomVal) randomDelay();
       return nextStepOk();
   }
}
```


Dettagli Di Circuit Breaker

- Le richieste di esecuzione del servizio arrivano alla classe `CircuitBreaker`, tramite chiamate al metodo `getResultBoundTime()` [potrebbero essere intercettate dal servizio `nextStep()` tramite un aspetto e redirezionate qui]
- Se il circuito è *chiuso*, tramite `supplyAsync()`, si avvia il servizio
- Quando si rileva che il risultato non è disponibile entro un tempo di 300 ms, tramite `get()` con timeout, oppure quando si cattura l'eccezione eventualmente lanciata, allora il servizio non sta eseguendo bene, quindi passa nello stato *aperto*
- Se il circuito è aperto, si resta nello stato di aperto per 200 ms, quindi si passa allo stato semi-aperto, e vi rimane per 100 ms. Nello stato di semi-aperto si fa passare la metà delle richieste. Ogni volta che arriva una richiesta si valuta se è necessario passare allo stato successivo, ovvero da aperto a semi-aperto, o da semi-aperto a chiuso

```java
public interface BreakerState {     // role State
   public BreakerState nextState();
   public boolean isClosed();
}


public class Closed implements BreakerState {   // role ConcreteState
   public Closed() {
      System.out.println("go to closed");
   }

   @Override
   public BreakerState nextState() {
      return this;
   }

   @Override
   public boolean isClosed() {
      return true;
   }
}
```



```java
public abstract class OpenState implements BreakerState {
   private long openTime;
   private long deltaTimeOpen;
   private String st;

   public OpenState(long delta, String name) {
      openTime = System.currentTimeMillis();
      deltaTimeOpen = delta;
      st = name;
      System.out.println("go to " + st);
   }

   /**
    * if enough time has passed, go to next state
    * @param s a Supplier giving the next state
    */
   public BreakerState nextState(Supplier<BreakerState> s) {
      long elapsed = System.currentTimeMillis() - openTime;
      System.out.print("time in " + st + " " + elapsed + " ms, ");
      if (elapsed < deltaTimeOpen) {
         System.out.println("stay " + st);
         return this;
      }
      return s.get();     // change CircuitBreaker's state
   }
}
```



```java
public class HalfOpen extends OpenState { // role ConcreteState
   public HalfOpen() {
      super(100, "half open");  // half open for 0.1 sec
   }

   @Override
   public BreakerState nextState() {
      return nextState(() -> new Closed()); // supplies next state
   }

   /**
    * returns true half of the times, statistically
    */
   @Override
   public boolean isClosed() {
      return (0 == new Random().nextInt(2));
   }
}

public class Open extends OpenState { // role ConcreteState
   public Open() {
      super(200, "open"); // open for 0.2 sec
   }

   @Override
   public BreakerState nextState() {
      return nextState(() -> new HalfOpen()); // supplies next state
   }

   @Override
   public boolean isClosed() {
      return false;
   }
}
```



```java
public class CircuitBreaker { // role Context
   private Task t;
   private BreakerState state = new Closed();      // initial CircuitBreaker's state
   private CompletableFuture<Integer> task;

   public CircuitBreaker() {  t = new Task();  }

   public int getResultBoundTime() {
      state = state.nextState();       // check for change of CircuitBreaker's state
      if (!state.isClosed()) return -1;          // request to service not performed
      System.out.println("CircuitBreaker executing task");
      task = CompletableFuture.supplyAsync(() -> {  // start async request
         try {     return t.nextStep();             // request a service
         } catch (Exception e) {             // capture exception thrown by service
            System.out.println("An exception occurred in Task");
            state = new Open();                    // change CircuitBreaker's state
            return -1;
         }
      });
      return getValueFromFuture();
   }

   private int getValueFromFuture() {
      int result = -1;
      try { result = task.get(300, TimeUnit.MILLISECONDS);   // wait up to 300 ms
      } catch (InterruptedException e) {
      } catch (ExecutionException e) {
      } catch (TimeoutException e) {             // if result is not ready
         System.out.println("A timeout occurred in getting the result");
         state = new Open();                       // change CircuitBreaker's state
      }
      return result;
   }
}
```

Dettagli Del Test

- Si chiama per 30 volte il servizio, attraverso il CircuitBreaker
- Fra una chiamata e la successiva si introduce un ritardo di massimo 0,1 s
- Il CircuitBreaker restituisce al chiamante un valore numerico da 0 a 5 che corrisponde al risultato dell'esecuzione del servizio, oppure -1 quando l'esecuzione non è andata a buon fine
- Si contano i valori diversi da -1 per determinare quante volte il servizio ha eseguito in modo normale


```
public class TestBreaker {
   public static void main(String[] args) {
      System.out.println("Calling method ");
      long c = callMethodGet();
      System.out.println("Number of successful calls " + c);
   }

   private static long callMethodGet() {
      CircuitBreaker brk = new CircuitBreaker();
      // call with 0.1 sec max intervals for 30 times
      long c = IntStream.rangeClosed(1, 30).map(i -> {
         delay01sec();
         return brk.getResultBoundTime();      // request service
      }).filter(v -> v > -1).count();          // count actual results
      return c;
   }

   private static void delay01sec() {
      try {
         Thread.sleep(new Random().nextInt(100));
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }
   }
}
```