Ingegneria del software
Elaborato

Oualid Hamdi

1 Introduzione

L'elaborato ha lo scopo di simulare e migliorare la gestione dei lavoratori nel Magazzino Amazon.

In particolare, i lavoratori possono essere aggregati in team diversi, oppure indipendenti. Le attività svolte dai lavoratori sono due: loader e picker.

- **Loader**: Scarico del pacco da un container, posizionamento del pacco in un'area di stoccaggio e caricamento delle informazioni relative all'ubicazione del pacco scaricato in una cpda attraversp un server di messaggistica ActiveMQ (piattaforma che implementa le specifiche di Java Message Service).
- **Picker**: Ricezione del messaggio dalla coda contenente le informazioni riguardanti il pacco, recupero del pacco dall'area di stoccaggio e infine spedizione del pacco stesso.

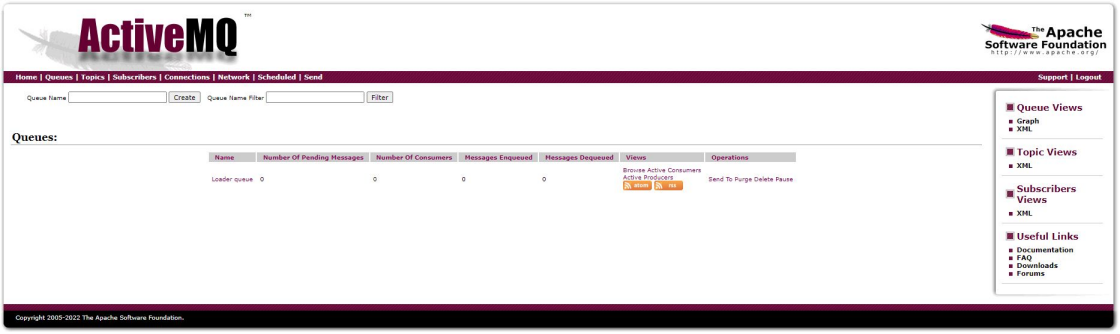
La gestione delle varie attività viene fatta attraverso un monitor che permette di registrare l'inattività del singolo e di cambiarne il ruolo tramite notifica.

2 Implementazione

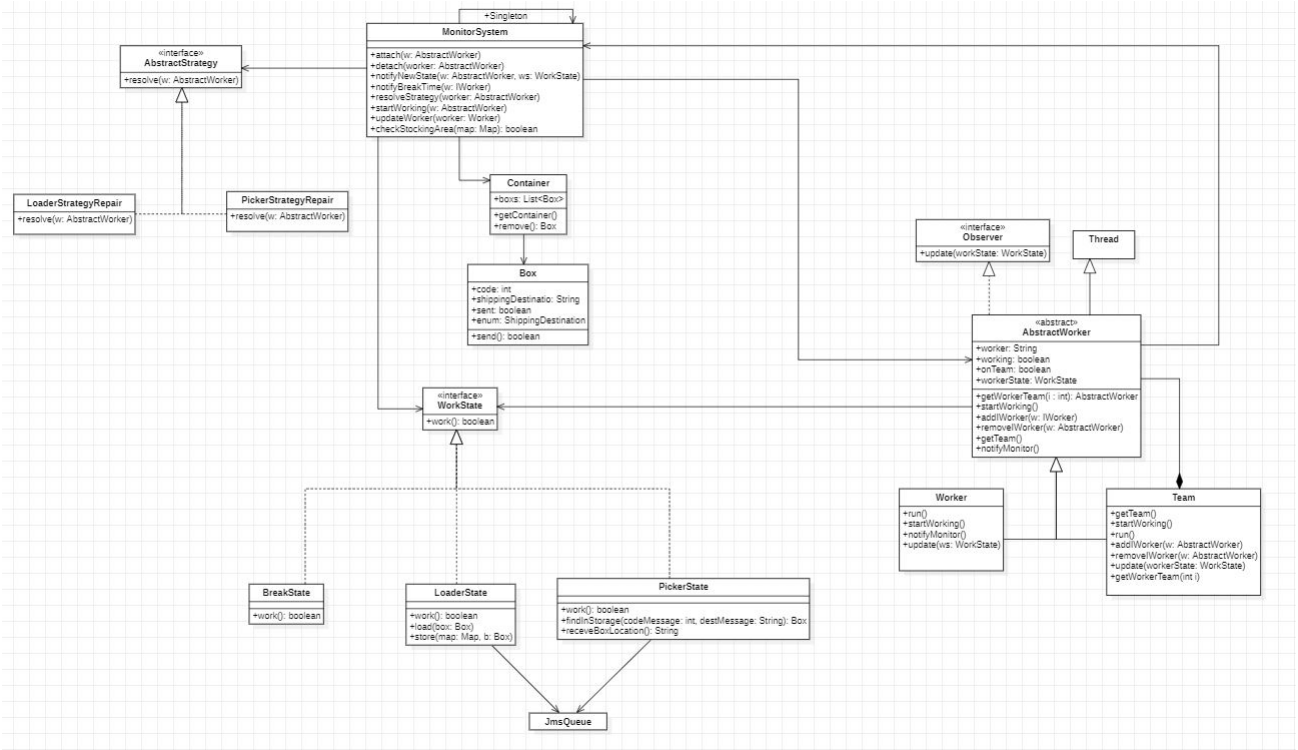
Per l'implementazione di questa simulazione, è stato utilizzato una coda JMS e i design pattern Composite, Observer, Strategy e State.

- **Composite**:permette di creare team di lavoratori rendendone più semplice la gestione. Vengono distinti perciò due figure diverse: Leaf, i lavoratori singoli, e Team, le aggregazioni di Lavoratori;
- **Observer**: gestisce a livello macroscopico il Monitor, ricevendo le notifiche di inattività o inviando ai lavorati la notifica del loro cambiamento di attività;
- **Strategy**: gestisce l'eventualità in cui il lavoratore non svolge alcuna attività, assegnandogliene una nuova.
- **State**: gestisce le attività possibili dei lavoratori all'interno del magazzino in modo efficace.

- **ActiveMQ** : JMS è un'interfaccia di programmazione che definisce come inviare e ricevere messaggi in un'architettura. Per implementarla è stato scelto il server di messaggistica ActiveMQ, una piattaforma open-source che fornisce un sistema di messaggistica robusto e scalabile per le applicazioni in Java.



2.1 Class Diagram



2.2 Discussione delle classi principali

- **AbstractWorker**

  La classe rappresenta il singolo lavoratore, questa oltre ad implementare l'interfaccia Observer estende la classe Thread.
  I metodi principali sono:

  1. addIWorker(AbstractWorker w) e removeIWormer(AbstractWorker w) per la composizione del team.

  2. startWorking() che attraverso il metodo run() dell'istanza di tipo Runnable segnala l'inizio del turno del lavoratore.

  3. notifyMonitor() per notificare alla classe MonitorSystem del cambiamento di stato del lavoratore.

- **MonitorSystem**

  La classa Monitorsystem è implementata come Singleton, richiesta solo tramite il metodo statico getMonitor().

  Il suo ruolo è quello di Observable per il design pattern Observer implementando i metodi attach(Observer), detach(Observer), notifyNewState(Observer w, WorkState ws) e notifyBreakTime(AbstractWorker w) per notificare i nuovi stati dei lavoratori.

  updateWorker(Worker worker) è un metodo fondamentale per la classe, in quanto permette di ottenere una strategia di risoluzione nel caso in cui i lavoratori non riescano a terminare la propria attività.

- **LoaderStratetegyRepair**
  La classe implementa l'interfaccia AbstractStrategy ed ha il metodo resolve(AbstractWorker w) che ha lo scopo di impostare lo stato, PickerState, al lavoratore permettendogli quindi di riprendere il lavoro con la nuova attività di Picker.

- **PickerStrategyRepair**
  La classe implementa l'interfaccia AbstractStrategy ed espone il metodo resolve(AbstractWorker w) che in base a dei controlli sul container e sull'area di stoccaggio può cambiare lo stato del lavoratore. Gli stati in cui possono essere cambiati sono LoaderState, PickerState o cessarne l'attività.

- **BreakState**
  La classe implementa l'interfaccia WorkState e rappresenta formalmente lo stato di break dei lavoratori, attraverso una momentanea inattività del Thread.

- **LoaderState**
  La classe implementa l'interfaccia WorkState e rappresenta il primo vero stato della catena di produzione del magazzino. Nel metodo principale work il lavoratore recupera un oggetto Box dal Container e lo posiziona nell'area di stocaggio apposita. Carica quindi un messaggio sulla Coda jms contenente le informazioni riguardo l'area di stoccaggio e il codice del pacco.

- **PickerState**
  La classe Implementa l'interfaccia WorkState e rappresenta la seconda fase della catena di produzione del magazzino. Nel metodo Work Il lavoratore riceve un messaggio dalla coda Jms per poter recuperare il pacchetto in zona di stoccaggio ed infine inviarlo. Il metodo work() ritorna false nel caso in cui non sia riuscito a completare la propria attività.

2.3 Codice e Testing

2.3.1 MonitorSystem

```
public class MonitorSystem {

   private static final Logger loggerApplication = LoggerFactory.getLogger("logApplication");

   private static MonitorSystem instance;
   public static List<Observer> workers = new ArrayList<>();
   private static Container container ;
   private static Map<String, List<Box>> map = new ConcurrentHashMap<>();

   private static WorkState loadState = new LoaderState();
   private static WorkState pickState = new PickerState();
   private static WorkState breakState = new BreakState();
   private static int number = 100;

   private static LoaderStratetegyRepair loadRepair = new LoaderStratetegyRepair();
   private static PickerStrategyRepair pickRepair = new PickerStrategyRepair();

   public MonitorSystem(int n) {
      MonitorSystem.container = new Container(n);
   }

   public static WorkState getLoadState() {
      return loadState;
   }

   public static synchronized Container getContainer() {
      return container;
   }

   public static Map<String, List<Box>> getMap() {
      return map;
   }

   public static WorkState getPickState() {
      return pickState;
   }

   public static MonitorSystem getMonitor() {
      if (instance == null)
         instance = new MonitorSystem(number);
      return instance;
   }

   public static void reset() {
       instance = new MonitorSystem(number);
   }

   public static void attach(Observer w) {
      workers.add(w);
   }

   public static void detach(Observer worker) {
      workers.remove(worker);
   }

   public void notifyNewState(Observer w, WorkState ws) {
      w.update(ws);

   }
```

```
   public void notifyBreakTime(AbstractWorker w) throws InterruptedException {
      WorkState tmpState = w.getWorkerState();
      w.update(breakState);
      Thread.sleep(1000);
      w.update(tmpState);
   }

   private void resolveStrategy(AbstractWorker worker) {
      if (worker.getWorkerState() == loadState)
         loadRepair.resolve(worker);
      else
         pickRepair.resolve(worker);
   }

   public void startWorking(AbstractWorker w) {
      if (w instanceof Worker && !w.isWorking()) {
         w.setWorkingState(true);
         loggerApplication.info(w.getWorkerName() + "sta iniziando a lavorare");
         w.start();
      }
      if (w instanceof Team) {
         for (int i = 0; i < w.getTeam().size(); i++)
            this.startWorking(w.getWorkerTeam(i));
      }

   }

   public void updateWorker(Worker worker) {
      loggerApplication.info("Il monitor e' stato notificato riguardo l'inattivita' di " +
           worker.getWorkerName());
      this.resolveStrategy(worker);

   }

   public static synchronized boolean checkStockingArea(Map<String, List<Box>> map) {
      for (Entry<String, List<Box>> entry : map.entrySet()) {
         if (!entry.getValue().isEmpty()) {
            return false;
         }
      }
      return true;
   }

}
```

2.3.2 AbstractWorker

```
    public abstract class AbstractWorker extends Thread implements Observer {

   protected String worker;
   private boolean working;
   protected boolean onTeam;
   protected MonitorSystem mr ;
   protected List<AbstractWorker> team;
   protected WorkState workerState;

   protected static final Logger loggerApplication = LoggerFactory.getLogger("logApplication");

   public WorkState getWorkerState() {
      return workerState;
   }
```

```
    public void setWorkerState(WorkState workerState) {
        this.workerState = workerState;
    }

    public boolean isWorking() {
        return working;
    }

    public void setWorkingState(boolean state) {
        this.working = state;
    }

    public String getWorkerName() {
        return worker;
    }

    public void setWorkerName(String iWorker) {
        this.worker = iWorker;
    }

    public void startWorking() {
    }

    public void addIWorker(AbstractWorker w) {
        throw new UnsupportedOperationException();
    }

    public void removeIWormer(AbstractWorker w) {
        throw new UnsupportedOperationException();
    }

    public AbstractWorker getWorkerTeam(int i) {
        throw new UnsupportedOperationException();
    }

    public List<AbstractWorker> getTeam() {
        throw new UnsupportedOperationException();
    }

    public void notifyMonitor() {
        throw new UnsupportedOperationException();
    }

}
```

2.3.3 Worker

```
   public class Worker extends AbstractWorker {

   public Worker(String workerName, WorkState workerState) {
      this.worker = workerName;
      this.workerState = workerState;
      this.onTeam = false;
   }

   @Override
   public void run() {
      while (!this.isInterrupted()) {
         if (this.isWorking()) {
            try {
               this.setWorkingState(this.getWorkerState().work());
               loggerApplication.info(this.getWorkerName() + " ha completato la sua attivita'");
            } catch (Exception e) {
               e.printStackTrace();
               this.setWorkingState(false);
            }
         } else
            notifyMonitor();
      }
   }


   @Override
   public void update(WorkState w) {
      this.setWorkerState(w);
      loggerApplication.info(this.getWorkerName() + " ha cambiato attivita'!");
   }

   @Override
   public void startWorking() throws IllegalThreadStateException {
      this.setWorkingState(true);
      loggerApplication.info(this.getWorkerName() + " inizia a lavorare");
      this.start();

   }

   @Override
   public void notifyMonitor() {
      MonitorSystem.getMonitor().updateWorker(this);

   }

}
```

2.3.4 Team

```
public class Team extends AbstractWorker {

public Team(String teamName, MonitorSystem mr, WorkState workerState) {
   this.worker = teamName;
   this.team = new ArrayList<>();
   this.workerState = workerState;
   this.mr = mr;
}

@Override
public void addIWorker(AbstractWorker w) {
   if (w.getWorkerState() == this.workerState) {
      if (w instanceof Worker && !w.onTeam) {
         w.onTeam = true;
         loggerApplication.info("Il lavoratore " + w.getWorkerName() + " e' stato aggiunto a
              questo team : "
               + this.getWorkerName());
         this.team.add(w);
         MonitorSystem.attach(w);
      }
      if (w instanceof Team) {
         loggerApplication.info(
               "Il team " + this.getWorkerName() + " vuole aggiungere i membiri del team : " +
                    w.getWorkerName());
         while(w.getTeam().size()>0) {
            w.getWorkerTeam(0).onTeam = false;
            MonitorSystem.detach(w);
            this.addIWorker(w.getWorkerTeam(0));
            w.getTeam().remove(0);
         }
      }
   } else {
      loggerApplication.info("Il lavoratore " + w.getWorkerName() + " ha un compito diverso
           dal team : "
            + this.getWorkerName() + " per aggiungerlo cambiare attivi al lavoratore o al team
                 !");
   }
}

@Override
public void removeIWormer(AbstractWorker w) {
   team.remove(w);
   MonitorSystem.detach(w);
   w.onTeam = false;
}

@Override
public List<AbstractWorker> getTeam() {
   return team;
}

@Override
public AbstractWorker getWorkerTeam(int i) throws UnsupportedOperationException {
   return team.get(i);
}
```

```
   @Override
   public void startWorking() {
      loggerApplication.info("il team " + this.getWorkerName() + " inizia a lavorare ..");
      for (int i = 0; i < this.getTeam().size(); i++) {
         if (this.getWorkerTeam(i) instanceof Worker && this.getWorkerTeam(i).interrupted())
            this.getWorkerTeam(i).startWorking();
         else
            this.getWorkerTeam(i).startWorking();
      }
   }

   @Override
   public void run() {
      for (AbstractWorker w : team)
         w.run();
   }

   @Override
   public void update(WorkState workerState) {
      for (int i = 0; i < team.size(); i++) {
         if (this.getWorkerTeam(i) instanceof Worker)
            this.getWorkerTeam(i).setWorkerState(workerState);
         else
            this.getWorkerTeam(i).update(workerState);
      }
   }

}
```

2.3.5 Observer

```
   public interface Observer {

    void update(WorkState workerState);

}
```

2.3.6 WorkState

```
   public interface WorkState {

   static final Logger loggerApplication = LoggerFactory.getLogger("logApplication");

   boolean work() throws Exception;
}
```

2.3.7 PickerState

```
public class PickerState implements WorkState {

   @Override
   public boolean work() throws Exception {
      try {
         Box box;
         String s = receveBoxLocation();
         if (s != null) {
            int codeMessage = Integer.parseInt(s.replaceAll("[^0-9]", ""));
            String destMessage = s.substring(s.indexOf(":") + 1, s.lastIndexOf("."));
            box = findInStorege(codeMessage, destMessage);
            while(box == null)
                box = findInStorege(codeMessage, destMessage);
            return box.send();
         }
      } catch (JMSException e) {
         loggerApplication.info(e.getMessage());
      } catch (Exception e) {
         loggerApplication.info(e.getMessage());
         throw new Exception();
      }
      return false;
   }

   private static Box findInStorege(int codeMessage, String destMessage){
      List<Box> objects = Collections.synchronizedList(MonitorSystem.getMap().get(destMessage));
      for (Box b : objects) {
         if (b.getCode() == codeMessage) {
            objects.remove(b);
            return b;
         }
      }
      loggerApplication.error("Il pacco con codice : " + codeMessage + " con direzione : " +
          destMessage
           + " non si trova nel magazzino ");
      return null;
   }

   private static String receveBoxLocation() throws JMSException {
      String url = ActiveMQConnection.DEFAULT_BROKER_URL;
      String queueName = "Loader queue";
      ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
      Connection connection = connectionFactory.createConnection();
      connection.start();
      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      Destination destination = session.createQueue(queueName);
      MessageConsumer consumer = session.createConsumer(destination);
      try {
         Message message = consumer.receive(500);
         if (message instanceof TextMessage) {
            TextMessage textMessage = (TextMessage) message;
            connection.close();
            return textMessage.getText();
         }
      } catch (JMSException e) {
         loggerApplication.error(e.getMessage());
         throw new JMSException("Error on Jms Connection");
      }
      connection.close();
      return null;
   }
}
```

2.3.8 LoaderState

```
public class LoaderState implements WorkState {

   public boolean work() throws Exception {
      try {
         Box box = MonitorSystem.getContainer().remove();
         if (box != null) {
            store(MonitorSystem.getMap(), box);
            loggerApplication.info("Un pacco  stato rimosso dal Container e caricato nella mappa");
            this.load(box);
            return true;
         } else {
            loggerApplication.info("Non ci sono pi pacchi da spedire");
            return false;
         }
      } catch (JMSException e) {
         loggerApplication.error(e.getMessage());
         throw new JMSException("Error on Jms Connection");
      }
   }

   private static synchronized void store(Map<String, List<Box>> map, Box b) {
      if (map.containsKey(b.getShippingVan())) {
         List<Box> boxDestination = map.get(b.getShippingVan());
         boxDestination.add(b);
         map.put(b.getShippingVan(), boxDestination);
      } else {
         List<Box> boxDestination = Collections.synchronizedList(new ArrayList<Box>());
         boxDestination.add(b);
         map.put(b.getShippingVan(), boxDestination);
         loggerApplication.info("Creata area di stoccaggio per destinazione : " +
              b.getShippingVan());
      }
   }

   public void load(Box box) throws JMSException {

      String url = ActiveMQConnection.DEFAULT_BROKER_URL;
      String queueName = "Loader queue";

      ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
      Connection connection = connectionFactory.createConnection();
      connection.start();

      Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
      Destination destination = session.createQueue(queueName);
      MessageProducer producer = session.createProducer(destination);
      TextMessage message = session
            .createTextMessage("Box code " + box.getCode() + " shipping to :" +
                 box.getShippingVan() + ".");
      producer.send(message);
      connection.close();
   }
}
```

2.3.9 BreakState

```
public class BreakState implements WorkState {

@Override
public boolean work() throws InterruptedException {
   try {
      loggerApplication.info("Break-time !");
      Thread.sleep(10000);
      loggerApplication.info("Back to work !");
      return true;
   } catch (InterruptedException e) {
      e.printStackTrace();
      throw new InterruptedException();
   }
}

}
```

2.3.10 AbstractStrategy

```
public interface AbstractStrategy {

static final Logger loggerApplication = LoggerFactory.getLogger("logApplication");

void resolve(AbstractWorker w);

}
```

2.3.11 PickerStrategyRepair

```
public class PickerStrategyRepair implements AbstractStrategy {

@Override
public void resolve(AbstractWorker w) {
   if (w instanceof Worker) {
      if (!MonitorSystem.getContainer().getListContainer().isEmpty()) {
         w.setWorkerState(MonitorSystem.getLoadState());
         w.setWorkingState(true);
         loggerApplication.info("Il lavoratore " + w.getWorkerName() + " si sposta in zona
              picking");
      } else if (MonitorSystem.checkStockingArea(MonitorSystem.getMap())) {
         w.setWorkingState(false);
         loggerApplication.info("Il lavoratore " + w.getWorkerName() + " ha terminato la sua
              attivita'");
         w.interrupt();
      } else {
         w.setWorkingState(true);
         loggerApplication.info("Il lavoratore " + w.getWorkerName() + " continua a lavorare in
              zona picking");
      }
   }
}

}
```

2.3.12 LoaderStratetegyRepair

```
public class LoaderStratetegyRepair implements AbstractStrategy {

@Override
public void resolve(AbstractWorker w) {
   if (w instanceof Worker)
      loggerApplication.info(
            "Il lavoratore " + w.getWorkerName() + " non avendo pi pacchi da scaricare, si
                 sposta in zona picking");
   w.setWorkerState(MonitorSystem.getPickState());
   w.setWorkingState(true);

}

}
```

2.3.13 Container

```
public class Container {

private List<Box> boxs;

public Container() {}

public Container(int n) {
   this.boxs = Collections.synchronizedList(new ArrayList<Box>());
   for (int i = 0; i < n; i++) {
      Box b = new Box();
      b.setCode(i);
      boxs.add(b);
   }
}

public List<Box> getListContainer() {
   return boxs;
}

public synchronized Box remove() {
   return (!boxs.isEmpty()) ? boxs.remove(0) : null;
}

}
```

2.3.14 Box

```
public class Box {

private int code;
private String shippingDestination;
private Boolean sent;

private static final Logger loggerApplication = LoggerFactory.getLogger("logApplication");

public Box() {
   this.setShippingVan(this.getDestination());
   this.sent = false;
}
```

```
   private static enum ShippingDestination {

      Firenze,
      Provincie,
      Prato,
      Siena,
      Pisa,
      Montecatini

   }

   public void setCode(int code) {
      this.code = code;
   }

   public int getCode() {
      return code;
   }

   public String getShippingVan() {
      return shippingDestination;
   }

   private void setShippingVan(String shippingVan) {
      this.shippingDestination = shippingVan;
   }

   private String getDestination() {
      int pick = new Random().nextInt(ShippingDestination.values().length);
      return ShippingDestination.values()[pick].toString();
   }

   public boolean send() {
      if (!sent) {
         loggerApplication.info("Il pacco  stato correttamente spedito");
         return this.sent = true;
      }

      loggerApplication.error("Il pacco  stato gi spedito");
      return false;

   }

}
```

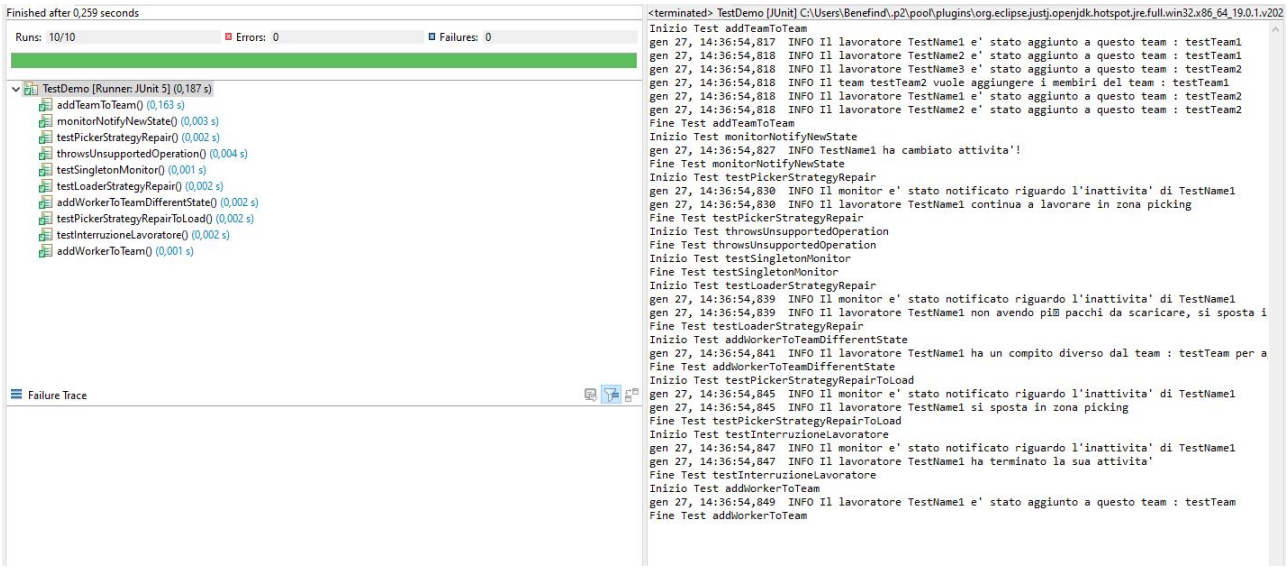
2.3.15 Classe di Test

```
public class TestDemo {

@BeforeEach
void setUp(){
   MonitorSystem.reset();
}

@Test
void throwsUnsupportedOperation() {
   System.out.println("Inizio Test throwsUnsupportedOperation");
   Team teamTest = new Team("testTeam", MonitorSystem.getMonitor(),
        MonitorSystem.getLoadState());
   Worker worker1 = new Worker("TestName1", MonitorSystem.getLoadState());
   Worker worker2 = new Worker("TestName2", MonitorSystem.getLoadState());
   assertThrows(UnsupportedOperationException.class, () -> worker1.addIWorker(worker2));
   assertThrows(UnsupportedOperationException.class, () -> worker1.getTeam());
   assertThrows(UnsupportedOperationException.class, () -> worker1.getWorkerTeam(0));
   assertThrows(UnsupportedOperationException.class, () -> worker1.removeIWormer(worker2));
   assertThrows(UnsupportedOperationException.class, () -> teamTest.notifyMonitor());
   System.out.println("Fine Test throwsUnsupportedOperation");
}

@Test
void testSingletonMonitor() {
   System.out.println("Inizio Test testSingletonMonitor");
   MonitorSystem mr1 = MonitorSystem.getMonitor();
   MonitorSystem mr2 = MonitorSystem.getMonitor();
   assertEquals(mr1, mr2);
   System.out.println("Fine Test testSingletonMonitor");
}

@Test
void testLoaderStrategyRepair() {
   System.out.println("Inizio Test testLoaderStrategyRepair");
   Worker worker1 = new Worker("TestName1", MonitorSystem.getLoadState());
   worker1.setWorkingState(false);
   worker1.notifyMonitor();
   assertTrue(worker1.isWorking());
   assertEquals(worker1.getWorkerState(), MonitorSystem.getPickState());
   System.out.println("Fine Test testLoaderStrategyRepair");
}

// Test resve strategy picker
// Caso in cui l'aread di stoccaggio  ancora piena.
@Test
void testPickerStrategyRepair() {
   System.out.println("Inizio Test testPickerStrategyRepair");
   Worker worker1 = new Worker("TestName1", MonitorSystem.getPickState());
   Box b = new Box();
   List<Box> test = new ArrayList<Box>();
   test.add(b);
   MonitorSystem.getMap().put("testKey", test);
   MonitorSystem.getContainer().getListContainer().clear();
   worker1.setWorkingState(false);
   worker1.notifyMonitor();
   assertTrue(worker1.isWorking());
   assertEquals(worker1.getWorkerState(), MonitorSystem.getPickState());
   System.out.println("Fine Test testPickerStrategyRepair");
}
```

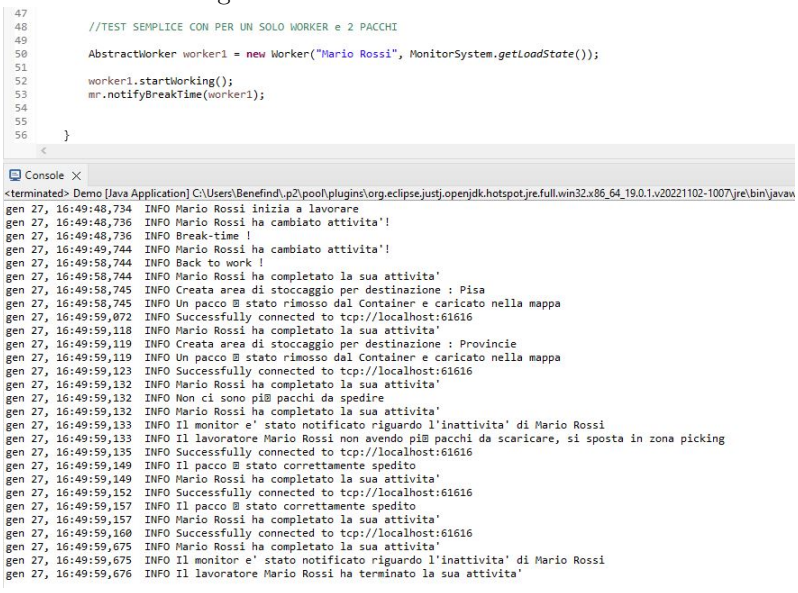
```
// Test picker che, a causa del Container non ancora vuoto si sposta in zona
// Loading.
@Test
void testPickerStrategyRepairToLoad() {
   System.out.println("Inizio Test testPickerStrategyRepairToLoad");
   MonitorSystem.reset();
   Worker worker1 = new Worker("TestName1", MonitorSystem.getPickState());
   worker1.setWorkingState(false);
   worker1.notifyMonitor();
   assertTrue(worker1.isWorking());
   assertEquals(worker1.getWorkerState(), MonitorSystem.getLoadState());
   System.out.println("Fine Test testPickerStrategyRepairToLoad");
}

@Test
void testInterruzioneLavoratore() {
   System.out.println("Inizio Test testInterruzioneLavoratore");
   Worker worker1 = new Worker("TestName1", MonitorSystem.getPickState());
   MonitorSystem.getContainer().getListContainer().clear();
   MonitorSystem.getMap().clear();
   worker1.setWorkingState(false);
   worker1.notifyMonitor();
   assertFalse(worker1.isWorking());
   System.out.println("Fine Test testInterruzioneLavoratore");
}

@Test
void addWorkerToTeam() {
   System.out.println("Inizio Test addWorkerToTeam");
   Team teamTest = new Team("testTeam", MonitorSystem.getMonitor(),
        MonitorSystem.getLoadState());
   Worker worker1 = new Worker("TestName1", MonitorSystem.getLoadState());
   teamTest.addIWorker(worker1);
   assertEquals(worker1, teamTest.getWorkerTeam(0));
   System.out.println("Fine Test addWorkerToTeam");
}

@Test
void addWorkerToTeamDifferentState() {
   System.out.println("Inizio Test addWorkerToTeamDifferentState");
   Team teamTest = new Team("testTeam", MonitorSystem.getMonitor(),
        MonitorSystem.getLoadState());
   Worker worker1 = new Worker("TestName1", MonitorSystem.getPickState());
   teamTest.addIWorker(worker1);
   assertThrows(IndexOutOfBoundsException.class, () -> teamTest.getWorkerTeam(0));
   System.out.println("Fine Test addWorkerToTeamDifferentState");
}

@Test
void addTeamToTeam() {
   System.out.println("Inizio Test addTeamToTeam");
   Team teamTest1 = new Team("testTeam1", MonitorSystem.getMonitor(),
        MonitorSystem.getLoadState());
   Team teamTest2 = new Team("testTeam2", MonitorSystem.getMonitor(),
        MonitorSystem.getLoadState());
   Worker worker1 = new Worker("TestName1", MonitorSystem.getLoadState());
   Worker worker2 = new Worker("TestName2", MonitorSystem.getLoadState());
   Worker worker3 = new Worker("TestName3", MonitorSystem.getLoadState());
   teamTest1.addIWorker(worker1);
   teamTest1.addIWorker(worker2);
   teamTest2.addIWorker(worker3);
   teamTest2.addIWorker(teamTest1);
   assertEquals(teamTest2.getTeam().size(), 3);
   System.out.println("Fine Test addTeamToTeam");
}
```

```
    @Test
    void monitorNotifyNewState() {
       System.out.println("Inizio Test monitorNotifyNewState");
       Worker worker1 = new Worker("TestName1", MonitorSystem.getLoadState());
       MonitorSystem.getMonitor().notifyNewState(worker1, MonitorSystem.getPickState());
       assertEquals(worker1.getWorkerState(), MonitorSystem.getPickState());
       System.out.println("Fine Test monitorNotifyNewState");

    }

}
```

Risultato Test JUnit

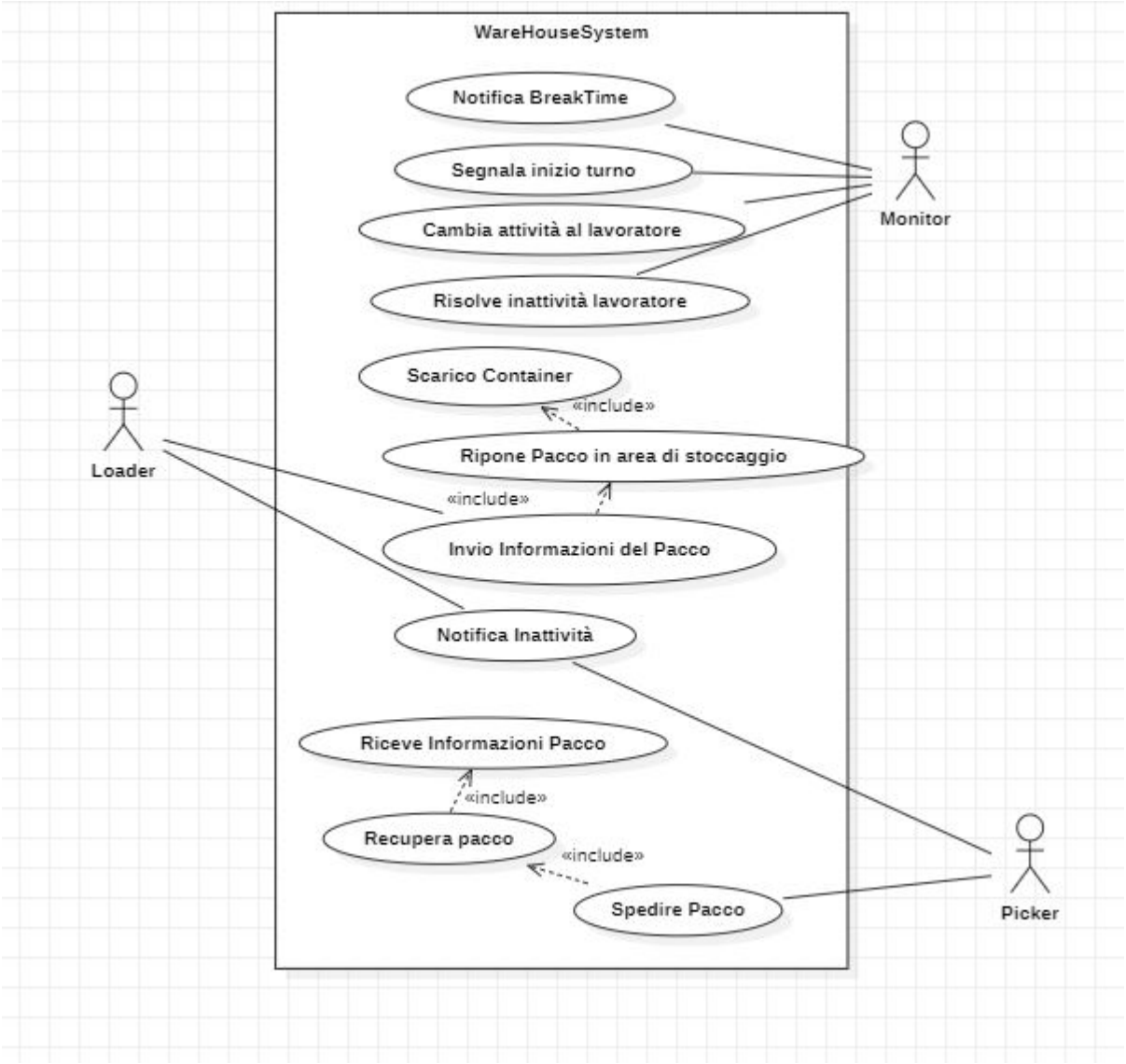


Console log un solo Worker



2.4 Use Case Diagram

Il seguente diagrama rappresenta i modi in cui gli attori, i lavoratori e il monitorSystem, interagiscono con il sistema e le funzionalità che questo fornisce.



2.5 Activity diagram

Il seguente diagrama mostra il flusso di attività del lavoratore. Partendo dallo startPoint il flusso si dirama in base allo stato del lavorate che tenta di esegurire la propria attività. Qualora non riesca a terminare l'attività, viene mostrato come il monitor decida se indirizzare il flusso alla ramificazione iniziale con un nuovo stato oppure termini il flusso.