COP5339 – Object Oriented Software Development

Written by:

Christopher Foley
Z15092976

Academic Year: 2016-2017

Table of Contents

Problem Definition

The following problem was posed.

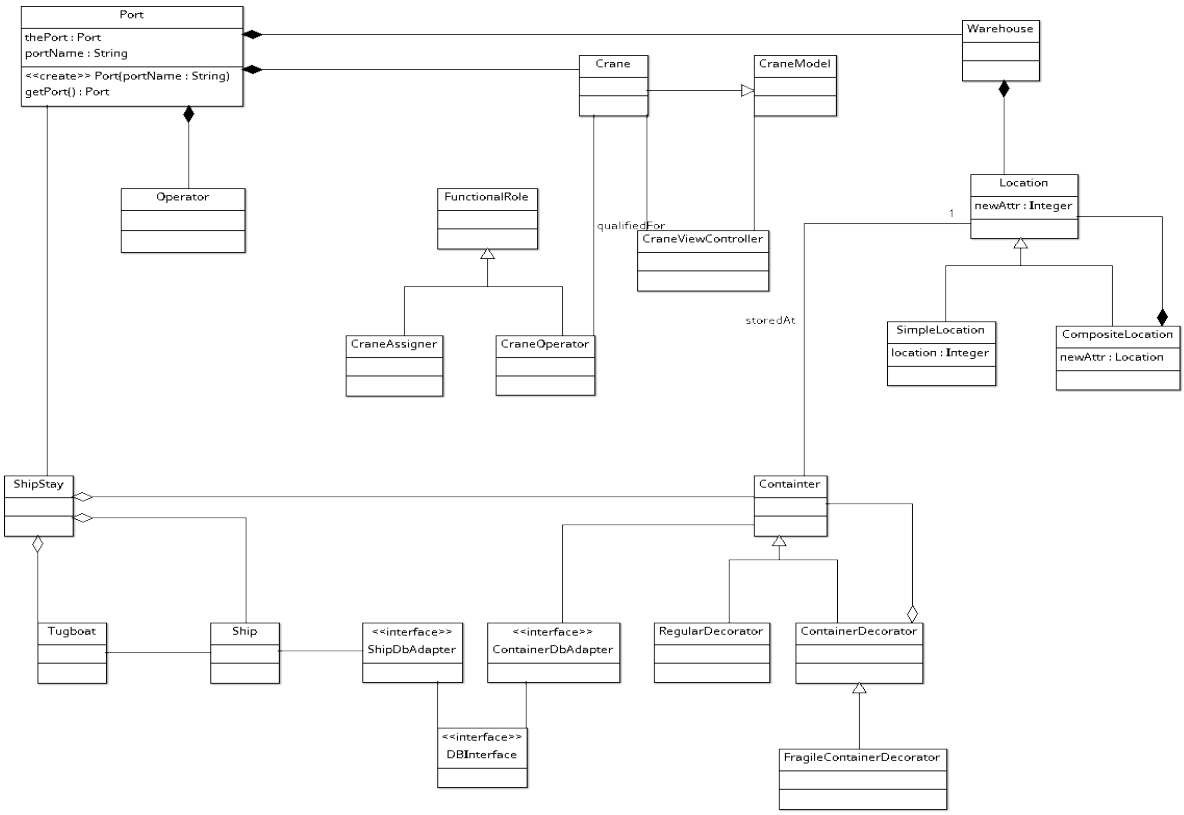
Consider the model of the port using patterns of Assignment 2 (attached).

- Q1) Write a Java or C++ program to transform Containers and Ships class interfaces into a relational database that has tables (relations): CARGO_UNIT, W_LOCATION, and VESSEL. Use an appropriate pattern in your code.

When a container arrives to the warehouse we must create a container object including all its relevant information (size, weight,…). We assign a location to this container. We then write this record to the database together with its corresponding location.

- Q2) Extend the UML class diagram of Q1 by: 1. Adding sensors to the warehouse locations and ships to detect when the crane has arrived. 2. Allowing different tugboats to be assigned to ships on arrival and departure. 3. Classifying cranes into Normal and Delicate Handling. The classes already in the model must not be changed, you can only add classes.

The following UML based map of a port was created as a starting point:



The Container class was created based upon the specifications from the following generic definition from Sea Box, Inc.:

Sea Box, Inc.®

CORPORATE OFFICE
802 Industrial Hwy.
East Riverton, NJ 08077-1910
Tel: (856) 303-1101
Fax: (856) 303-1501
www.seabox.com
e-mail: sales@seabox.com

Double Doors One Ends
8' X 8'6" Dry Freight ISO Cargo Container

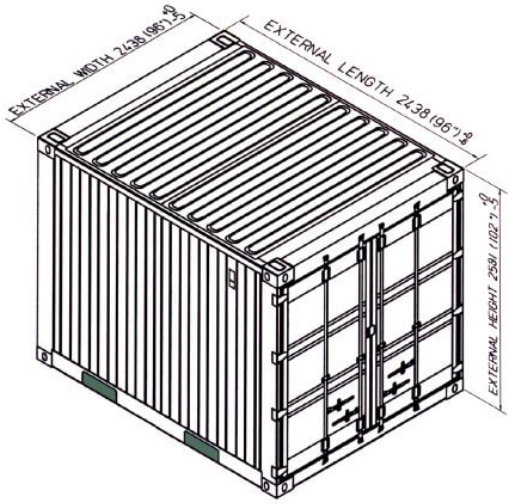
8' Dry Freight	Length		Height		Width		Door Opening	
	Exterior	Interior	Exterior	Interior	Exterior	Interior	Height	Width
Feet/Inches	8' 0"	7' 5 3/4"	8' 6"	7' 10 1/4"	8' 0"	7' 8 17/32"	7' 5 11/16"	7' 8 11/64"
Metric	2,438	2,280	2,591	2,394	2,438	2,350	2,278	2,341

8' Dry Freight	Tare Weight	Payload	Gross Weight	Cubic Capacity
Lbs.	3,150	18,895	22,045	453 CU.FT.
Kg.	1,429	8,571	10,000	12.8 CU.M.

ALL NEW CONTAINERS ARE MANUFACTURED TO THE LATEST ISO STANDARDS.

— STANDARD FEATURES —
- Corrugated steel sides and roof
- 14 gauge locking steel double end swing doors
- 1C " thick marine wood floors, forklift tested to 16,000 lbs per 44 square inches
- Tie down steel lashing rings, 4000 lbs. cap. each
- (2) way fork lift pockets
- Vents, (2) each

— OPTIONAL FEATURES —
- Restraint system designed to support 2" x 6" lumber
- Heavy duty steel tread plate floor
- Manifest box (2) each
- Adjustable shelving brackets (4) sets
- Decking and shoring beams (16) each



PART # SB830.6

8306

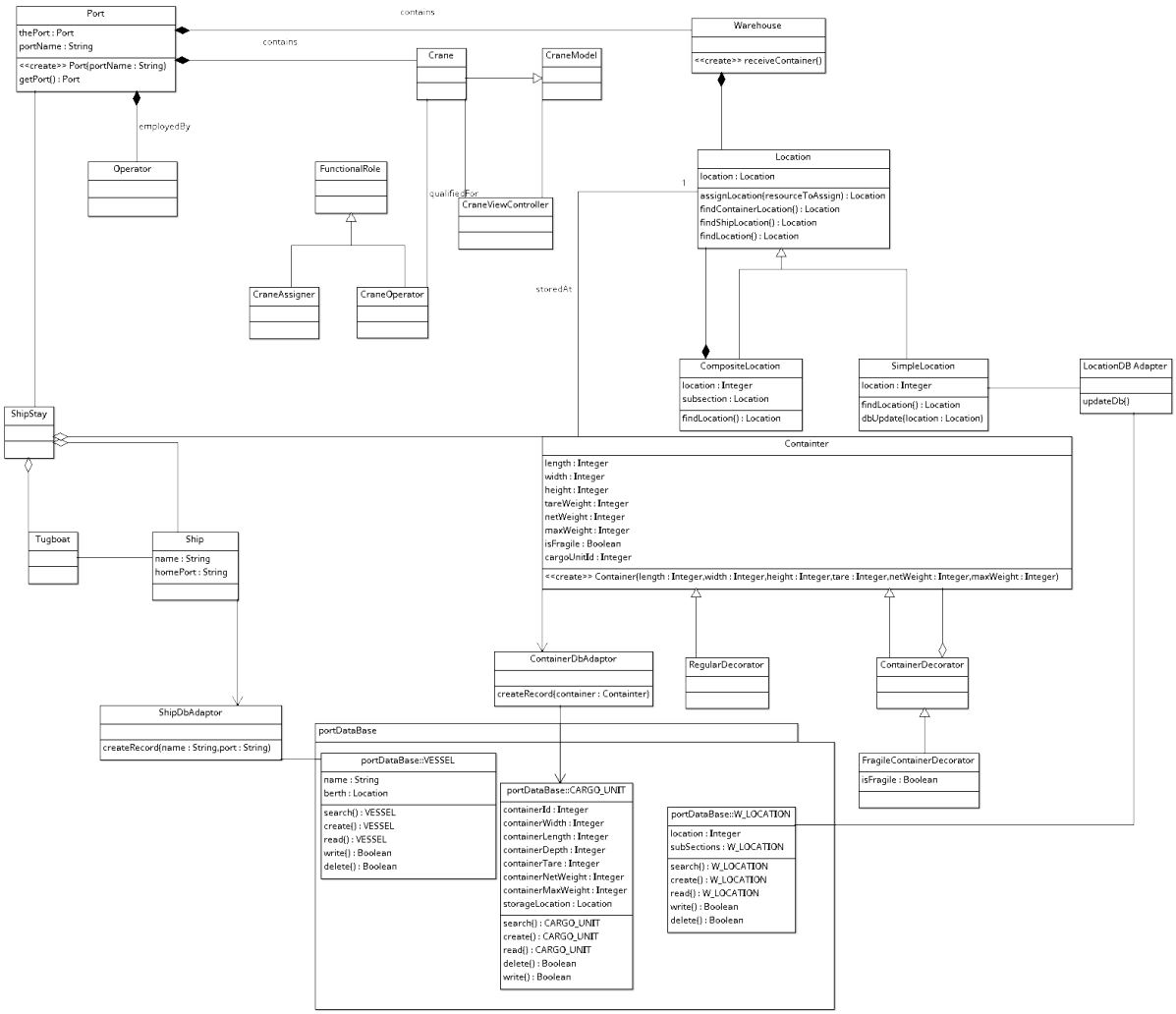
Container dimensions and weight are set to the exterior dimensions and defaulted to net weight,

handling is container specific. Only two types of containers are presently handled non specific or fragile.

Ship’s are identified by their name and registered home port, as per USCG recommendations.

Question 1: Transform Container and Ship Class

The Adapter Pattern was used to adapt the Container, Ship and Location classes to the data base, represented by the appropriate packages and classes. This adds an additional layer to the interface allowing the Container and Ship Classes to easily interface with the data base without extensive knowledge of the data base mechanisms.



Using ArgoUML to generate the Java code, this time with only 1 crash, resulted in code similar to the following:

ShipDBAdapter

```
import portDataBase;
public class ShipDbAdaptor {
  /* {author=Christopher Foley}*/



    public static portDataBase.VESSEL  myVESSELDB;

  public void createRecord(String name, String port) {
        myVESSELDB.create(name, port);
  }

  public void findShip(String name, String homePort) {
        myVESSELDB.search(name, port);
  }

  public void readShipRecord() {
        myVESSELDB.read();
 }


  public void writeShipRecords() {
        myVESSELDB.write();
 }

  public void deleteShipRecord() {
        myVESSELDB.delete();
 }

}
```

LocationDBAdapter

```
import java.util.Vector;
import portDataBase.W_LOCATION;

public class LocationDBAdapter {
  /* {author=Christopher Foley}*/
```

```
       private LocationDBAdapter() {
        }

  public void updateDb() {
  /* {author=Christopher Foley}*/
        W_LOCATION.write();          // write what you need to
  }



  public SimpleLocation findLocation(Integer searchLocation) {
        W_LOCATION.search();

  return null;
  }

}
```

ContainerDBAdapter

```
import portDataBase.CARGO_UNIT;

public class ContainerDbAdaptor {
      /* {author=Christopher Foley}*/


      public Vector  myCARGO_UNIT;

      public void createRecord(Containter container) {
            myCARGO_UNIT.create();
      }

      public Container locate()
      {
            myCARGO_UNIT.search();
            return null
      }

      public Container get()
      {
            myCARGO_UNIT.read();
      }

      public boolean delete()
      {
            return myCARGO_UNIT.delete();
      }

      public boolean updateRecord()
```

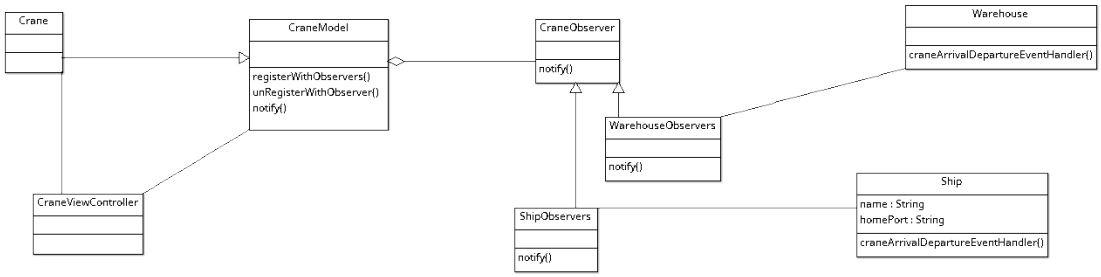
```
	{
		return myCargoUnit.write();
	}
}
```

Question 2

Q2) Extend the UML class diagram of Q1 by: 1. Adding sensors to the warehouse locations and ships to detect when the crane has arrived. 2. Allowing different tugboats to be assigned to ships on arrival and departure. 3. Classifying cranes into Normal and Delicate Handling. The classes already in the model must not be changed, you can only add classes.

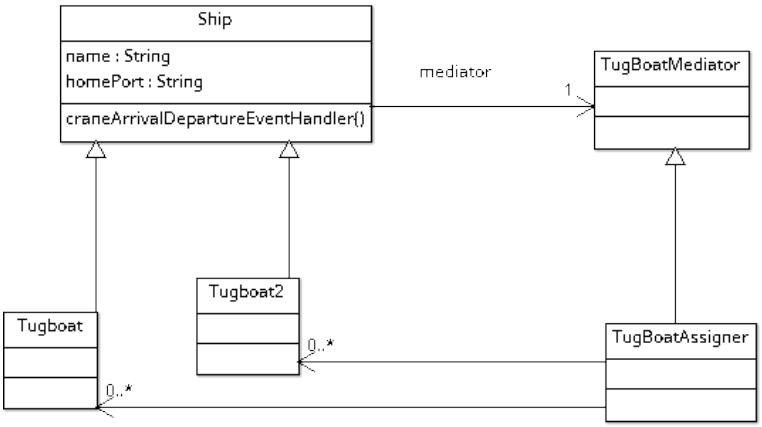
2.1 – Add Sensors

Adding sensors to the warehouse and crane is accomplished by the Observer class. This results in a sub diagram similar to the following:



2.2 – Assign Different tugboats

Different tugboats may be assigned to the ships. This is accomplished through use of the mediator pattern which decouples the tugs from the ships and permits different tugs to be assigned to one or many ships, thus permitting a large ship (“Queen Mary” or Aircraft Carrier) to be assigned different tugs depending on its weight and cargo (empty or full) and weather conditions. At a given moment 0..* tugboats may be assigned to a ship, this permits tug boats to be reassigned when a ship is berthed. This results in a sub diagram similar to the following:



2.3 – Classify Cranes as Normal and Delicate

Classification of Cranes as Normal and Delicate Handling can be accomplished through inheritance. Normal cranes will be classified as cranes, but delicate cranes may be classed as inheriting from cranes and their operations controlled or restricted via OCL. While wasteful, it restricts normal cranes from handling fragile materials unless the handling by normal cranes is restricted via the mechanisms of Assignment 3 (not included although it would be restricted by different sets of OCL).

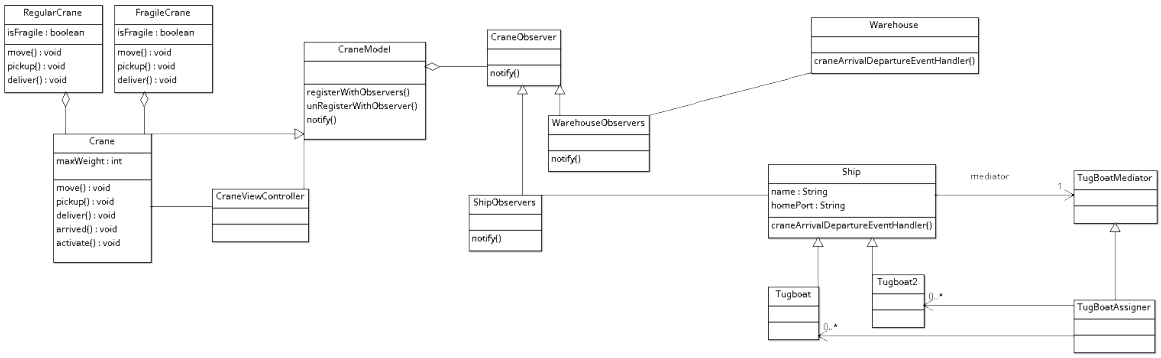
The OCL statements used for the FragileCrane are as follows:

- context FragileCrane::move () pre moveFragile: isFragile = true
- context FragileCrane::pickup () pre fragilePickup: isFragile = true
- context FragileCrane::deliver () pre fragileDeliver: isFragile = true

The OCL for the regularCrane were as follows:

- context RegularCrane::move () pre craneMove: isFragile = false
- context RegularCrane::pickup () pre cranePickup: isFragile = false
- context RegularCrane::deliver () pre craneDeliver: isFragile = false

This resulted in the following diagram:



The OCL resulted in code similar to the following, calls to the superclass for the regular crane were added to the code generated by argoUML.

RegularCrane.java

```
public class RegularCrane extends Crane {

  private final boolean isFragile = false;

   public RegularCrane()
   {
         super(30400);        // initialize to ISO intermodal maximum;
   }
    /**
   *
   * @precondition craneMove: isFragile = false
   */
  public void move() {
        super.move();
  }

    /**
```

```
   *
   * @precondition cranePickup: isFragile = false
   */
  public void pickup() {
        super.pickup();
  }

    /**
   *
   * @precondition craneDeliver: isFragile = false
   */
  public void deliver() {
        super.deliver();
  }

}
```

FragileCrane.java

```
public class FragileCrane extends Crane {

  private final boolean isFragile = true;

    public FragileCrane()
    {
      super(15200);           // initialize to 1/2 intermodal weight
    }
    /**
   *
   * @precondition moveFragile: isFragile = true
   */
  public void move() {
        super.move();
  }

    /**
   *
   * @precondition fragilePickup: isFragile = true
   */
  public void pickup() {
        super.pickup();
  }

    /**
   *
   * @precondition fragileDeliver: isFragile = true
   */
  public void deliver() {
        super.deliver();
  }

}
```

Crane.java

```
public class Crane extends CraneModel {
  /* {author=Christopher Foley}*/


  public final int maxWeight;

  public Crane(int weight)
  {
        maxWeight = weight;
  }

  public void move() {
  }
  public void pickup() {
  }
  public void deliver() {
  }
  public void arrived() {
  }
  public void activate() {
  }
}
```