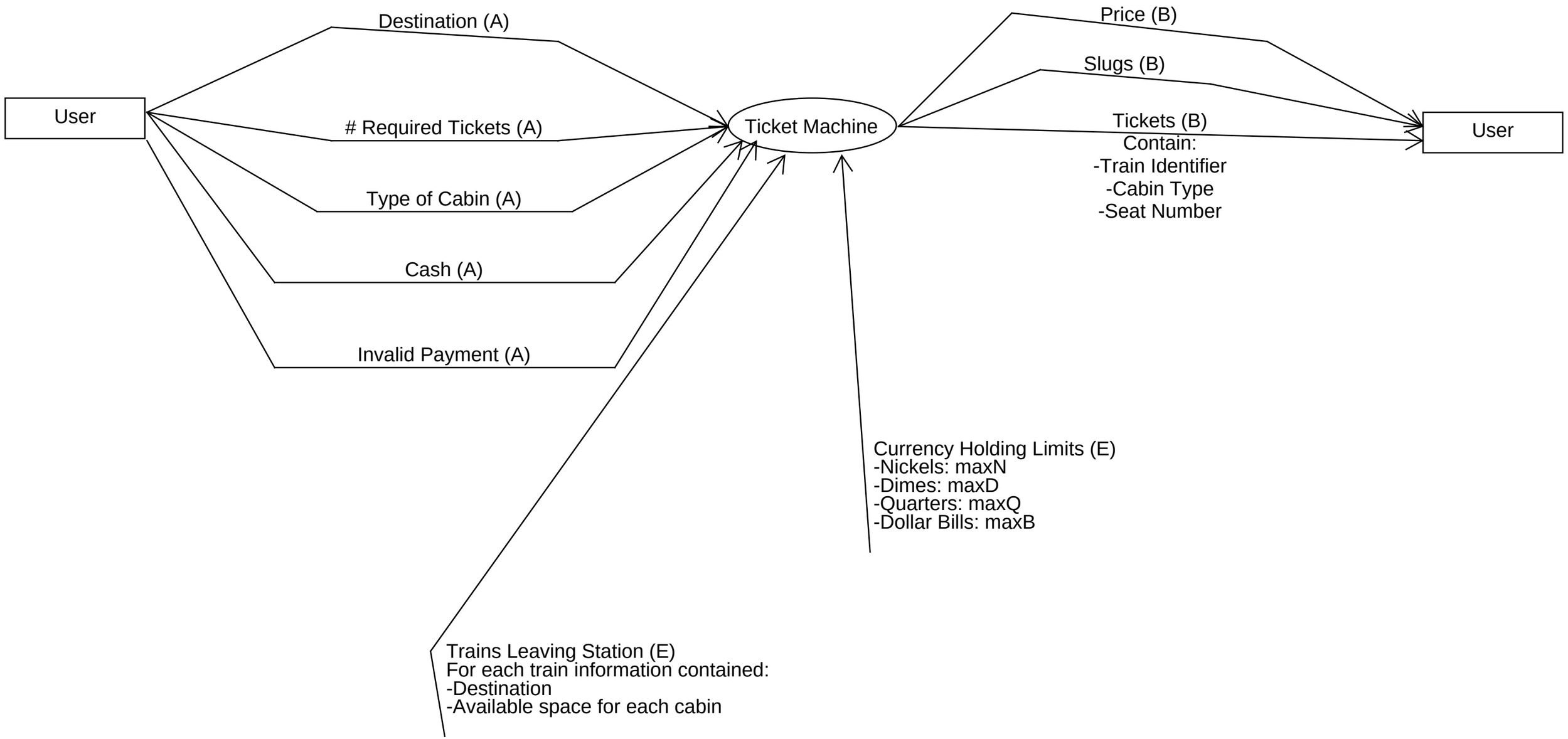
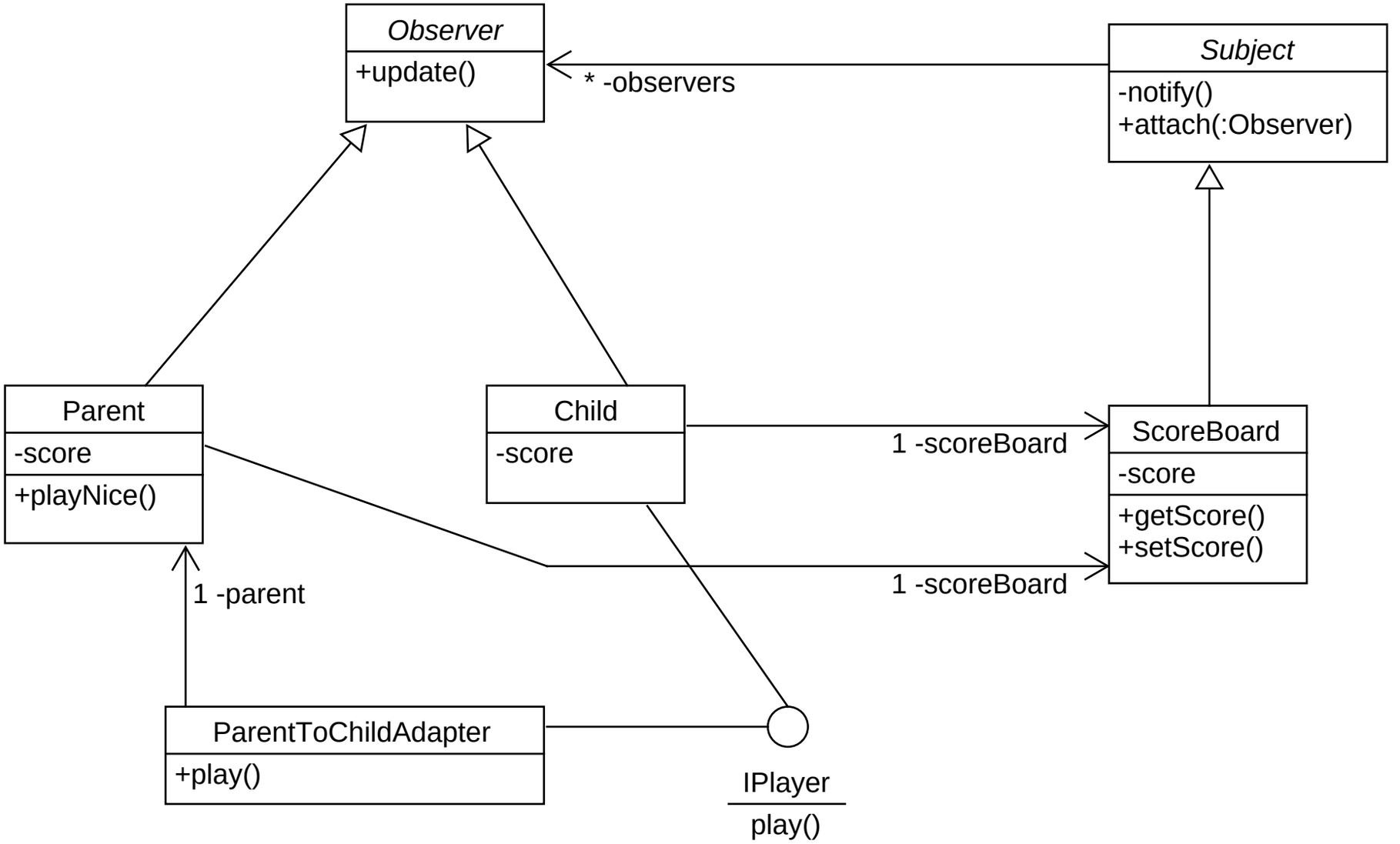
Karl Molina, Dana Parker	**Esof 322 HW 3**					
Train Station Function Points						
Reliable Backup and recovery	F1	0				
Data communications	F2	5				
Distributed Functions	F3	2				
performance	F4	3				
heavily used configuration	F5	1				
online data entry	F6	5				
operation ease	F7	5				
online update	F8	5				
complex interface	F9	0				
complex processing	F10	0				
reusability	F11	1				
installation ease	F12	0				
multiple sites	F13	4				
facilitate change	F14	3				
SUM	=	34				
TCF = 0.65 + .01 Sum of above	=	0.99				
	Complexity Weights					
Description	Low	Medium	High		totals	weighted
External Inputs (A)	3	4	6		5	20
External Outputs (B)	4	5	7		3	15
External Inqueries (C)	3	4	6		0	0
External Files (D)	7	10	15		0	0
Internal Files (E)	5	7	10		2	14
					UFC= Sum of weighte	49
FP = UFC x TCF =	48.51					

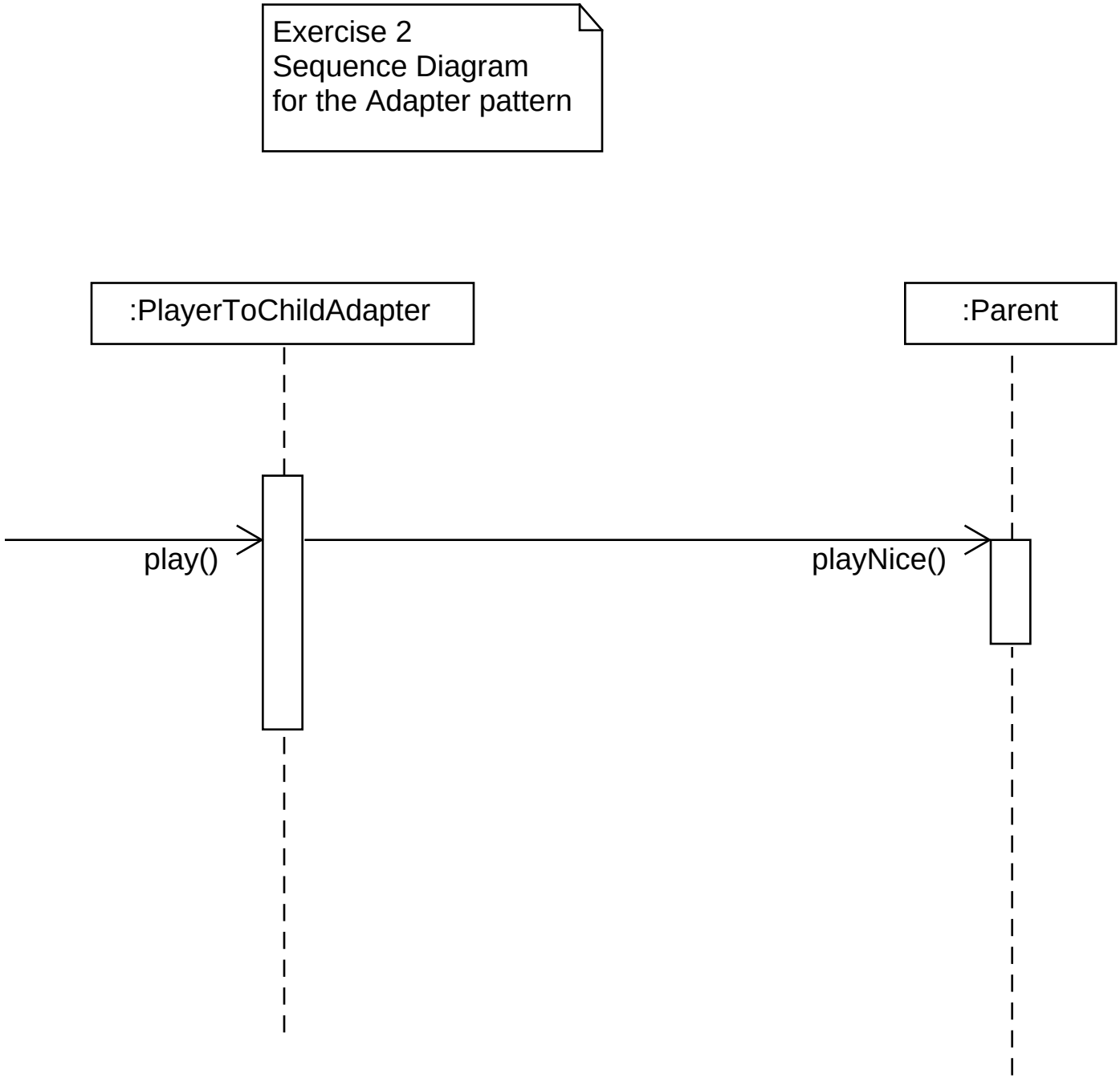
Exercise 1
Flow Diagram of
using a ticket machine
at a train station

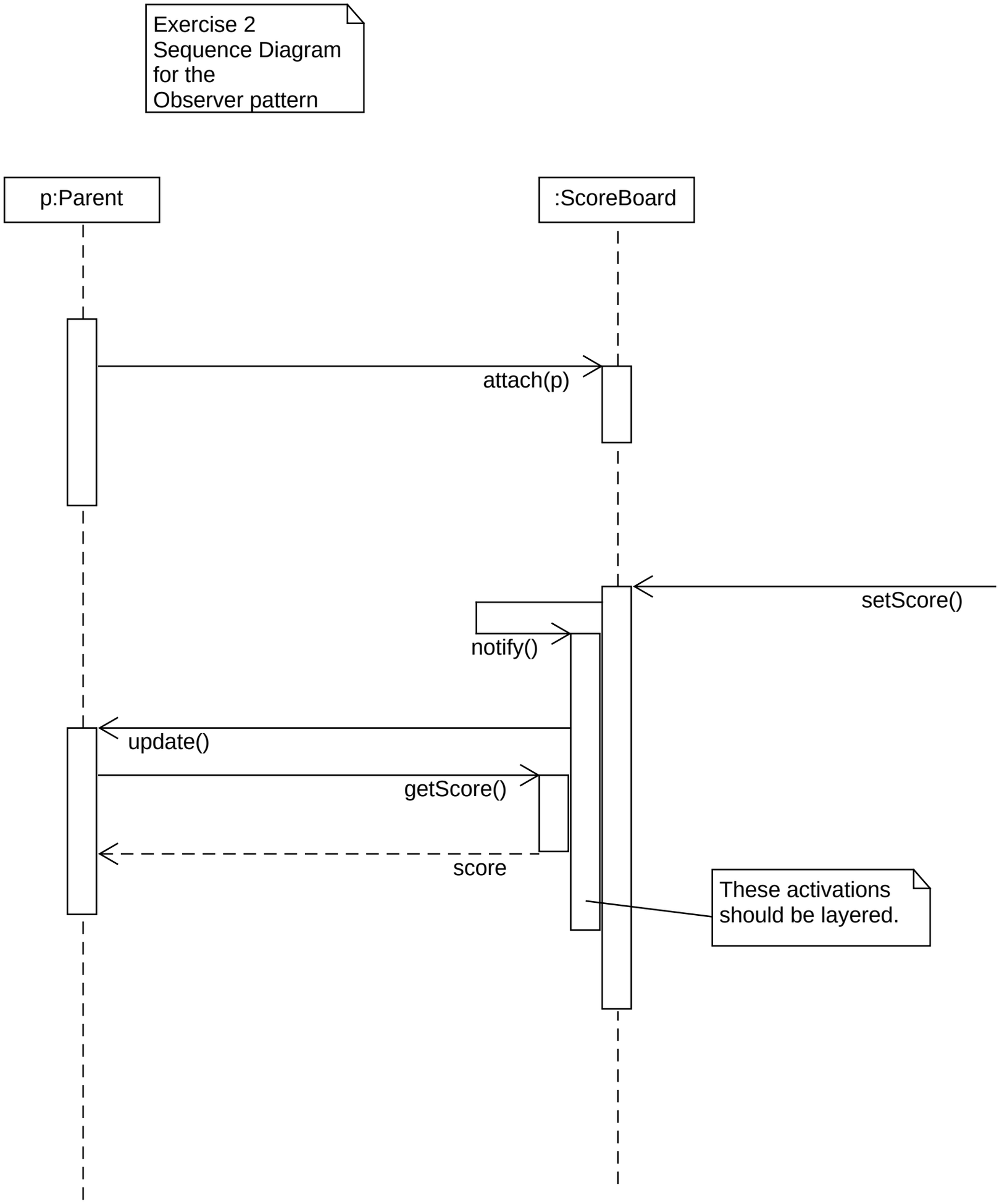


Exercise 2
Showing the coupling
of the Observer pattern
and the Adapter pattern.

Scenario:
At a game there are
children and parents.
They can both see
the score but usually
only children can be
players. This time a
parent wants to play.




Exercise 2
Sequence Diagram
for the Adapter pattern
:PlayerToChildAdapter
:Parent
play()
playNice()


Exercise 2
Sequence Diagram
for the
Observer pattern
p:Parent
:ScoreBoard
attach(p)
setScore()
notify()
update()
getScore()
score
These activations
should be layered.

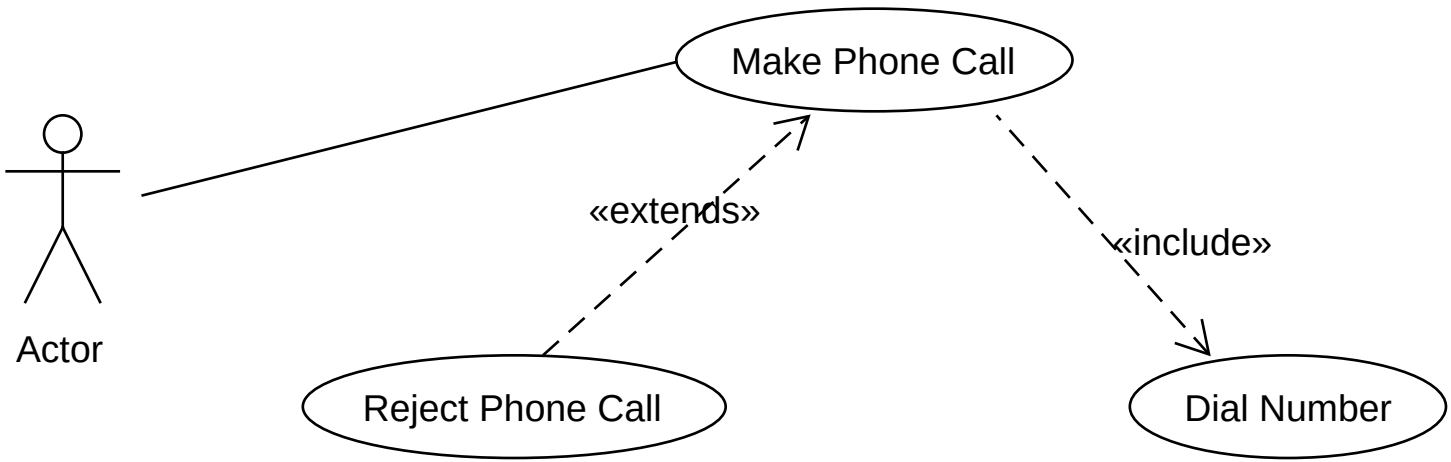
Exercise 3

Includes - If a base case includes another use case, then the other use case needs to happen in order for the base case to happen. In our example, you must dial the number in order to make the phone call.

Extends - If a use case extends a base case, then the use case may or may not happen. The base case doesn't need to be completed. In our example, the phone call may or may not be rejected.

Sources : Hamilton, Miles Learning UML 2.0 2006

Exercise 3
Use case diagram of making a phone call



Exercise 4
UML Class Diagram Relationships ordered by refactoring complexity. Lowest (1) is easiest to refactor and Highest (6) is the hardest to refactor.

1. Dependency
2. Association
3. Aggregation
4. Composition
5. Specialization
6. Inheritance

```
Class A {
        B b;

        Void method() {
                C c;
        }

}
```

We believe dependency is easier to refactor than association because with a dependency you would only have to refactor that method, instead of an association where all the references to the attribute would need to be updated. For example, the Class A above has an association with class B. If you changed the class B you would have to update all the code using A.b. You would have to find each place where this happens. However with the dependency that class A has on class B you only need to change that single method.

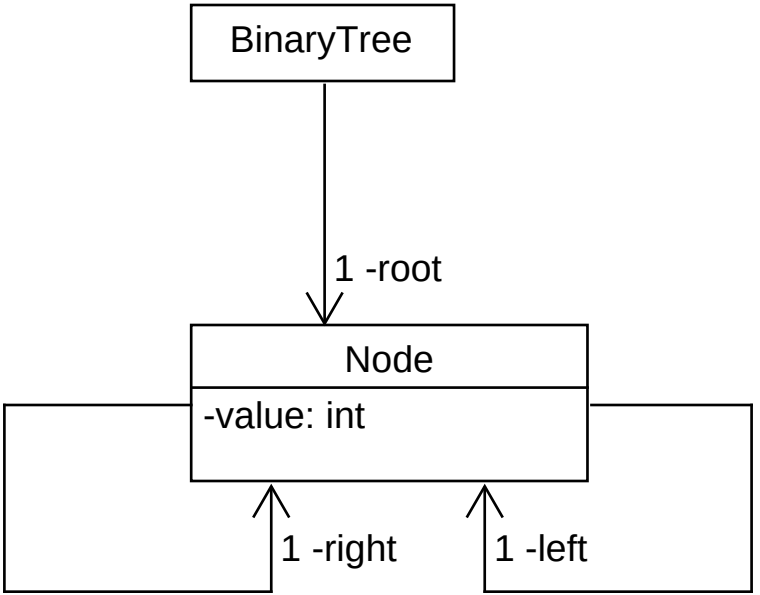
Dependency and association are easier to refactor than aggregation and composition because aggregation and composition are more complex versions of associations. With aggregation the association means that something is part of a whole but the whole can exist without the parts. If you refactored an aggregation you would not only have to update the references to the attribute but also update the structure of the relationship so that the “part” of the whole doesn’t get removed with the “whole.”

Composition is harder to refactor than aggregation because there is another relationship that must be met in the refactoring. You must make sure that if the “whole” is destroyed then each “part” is also destroyed.

Specialization and Inheritance are harder to refactor than composition and aggregation because if you changed the “whole” of the composition you wouldn’t necessarily have to change each “part’s” attributes and functions. But if you changed an interface then you would have to refactor every interface or class that implement that interface. This is not always possible because if you released a package to a customer that contained an interface, they may implement that interface in their own classes. If you wanted to change the interface you cannot go into their code and implement it properly. You would have to make parts of the interface obsolete and add new functions.

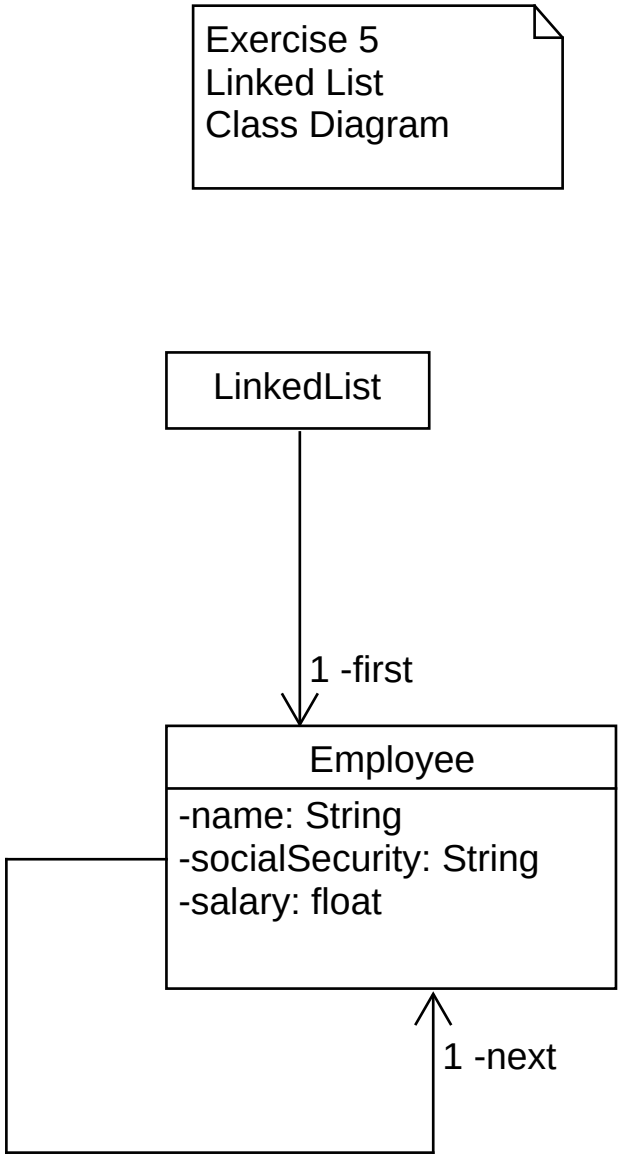
Inheritance is more difficult to refactor then specialization because now you can change more. A superclass may have function signatures but also implemented methods and also attributes. Changing a superclass will again cause you to change all the classes implementing it. This again might not be possible. However now you have to change more than just the functions, you have to update any code that used functions implemented in the superclass and update attributes and all of their references.

Exercise 5
Binary Tree
Class Diagram



```java
package esof322hw3;

/**
 *  ESOF 322 HW 3
 * @author Karl Molina, Dana Parker
 */
public class BinaryTree {
    private Node root;

    public BinaryTree(Node root) {
        this.root = root;
    }

    /**
     * Adds the node to the tree
     * @param n
     */
    public void addNode(Node n) {
        addNode(n, root);
    }

    /**
     * Recursive helper function to add the node to the correct spot in the tree
     * @param n
     * @param current
     */
    private void addNode(Node n, Node current) {
        if (n.getValue() < current.getValue()) {
            if (current.getLeft() == null) {
                current.setLeft(n);
            } else {
                addNode(n, current.getLeft());
            }
        } else {
            if (current.getRight() == null) {
                current.setRight(n);
            } else {
                addNode(n, current.getRight());
            }
        }
    }
}

/**
 * Node class that holds a value and left and right children
 * @author h89q624
 */
class Node {
    private Node left, right;
    private int value;

    /**
     * Initializes a Node with the value
     * @param value
     */
    public Node(int value) {
        this.value = value;
    }

    /**
     * Gets the node's value
     * @return
     */
    public int getValue() {
        return value;
    }

    /**
     * Gets the node's left child
```

```
70          * @return
71          */
72         public Node getLeft() {
73             return left;
74         }
75 
76         /**
77          * Gets the node's right child
78          * @return
79          */
80         public Node getRight() {
81             return right;
82         }
83 
84         /**
85          * Sets the node's right child
86          * @param n
87          */
88         public void setRight(Node n) {
89             right = n;
90         }
91 
92         /**
93          * Sets the node's left child
94          * @param n
95          */
96         public void setLeft(Node n) {
97             left = n;
98         }
99     }
```


Exercise 5
Linked List
Class Diagram
LinkedList
1 -first
Employee
-name: String
-socialSecurity: String
-salary: float
1 -next

```java
package esof322hw3;


/**
 *  ESOF 322 HW 3
 * @author Karl Molina, Dana Parker
 */
public class LinkedList {
    private Employee first;

    /**
     * Adds an employee to the linked list
     * @param e employee to add
     */
    public void add(Employee e) {
        if (first == null) {
            first = e;
        } else {
            e.setNext(first);
            first = e;
        }
    }

    /**
     * Gets the first employee in the linked list
     * @return
     */
    public Employee getFirst() {
        return first;
    }
}

class Employee {
    private String name, socialSecurity;
    private float salary;
    private Employee next;

    /**
     * Employee constructor
     * @param name name of employee
     * @param socialSecurity employee's social security number
     * @param salary employee's salary
     */
    public Employee(String name, String socialSecurity, float salary) {
        this.name = name;
        this.socialSecurity = socialSecurity;
        this.salary = salary;
    }

    /**
     * Gets the name of the employee
     * @return
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the social security number of the employee
     * @return
     */
    public String getSocialSecurity() {
        return socialSecurity;
    }

    /**
     * Gets the salary of the employee
     * @return
     */
```

```
70      public float getSalary() {
71          return salary;
72      }
73
74      /**
75       * Sets the next employee for the linked list
76       * @param e
77       */
78      public void setNext(Employee e) {
79          next = e;
80      }
81
82      /**
83       * Gets the next employee
84       * @return
85       */
86      public Employee getNext() {
87          return next;
88      }
89  }
```