Aspect-Oriented Smart Contracts

Refactoring to Role Objects

Saifur Rahman Bhuiyan

TU Dresden

October 14, 2022

Table of Contents

Introduction

Introduction

- Object-oriented system is typically based on a set of key abstractions. Each key abstraction is modeled by a corresponding class in terms of abstract state and behavior.
- However, once we want to scale up the system into an integrated suite of applications, we have to deal with different clients that need context-specific views on our key abstractions.
- Suppose we are developing software support for an organization. One of the key abstractions to be expressed is the concept of Person. Thus, our design model will include a Person class. The class provides operations to manage properties like the Person's name, address etc.

Introduction Cont.

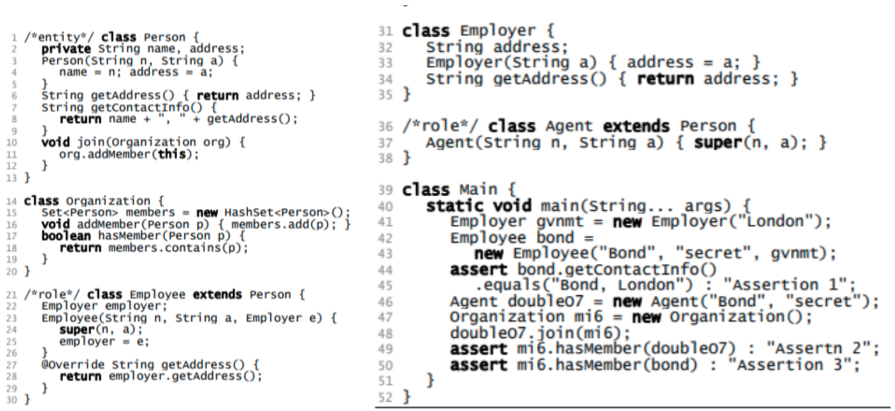
- Suddenly the organization needs software support. It seems our class design is inadequate to deal with a Person acting as Employee, Agent etc.
- Integrating several context-specific views in the same class will most likely lead to key abstractions with bloated interfaces.
- Such interfaces are difficult to understand and hard to maintain. Unanticipated changes cannot be handled gracefully and will trigger lots of recompilation.

Solution One

Roles as Sub-Classes

- A simple solution might be to extend the Persons class by adding new sub classes like Employee and Agent which will capture the Employee-specific and Agent-specific aspects respectively.
- However, while this implementation may seem plausible at first glance (Employee adds certain, employee-specific properties to Person), it does not scale.
- As soon as the second role is added to Person and the same entity must assume both roles (e.g., a person is both an employee and an agent), this approach hits the wall, since an object cannot be an instance of two sibling classes.

Sample Code



```java
/*entity*/ class Person {
   private String name, address;
   Person(String n, String a) {
      name = n; address = a;
   }
   String getAddress() { return address; }
   String getContactInfo() {
      return name + ", " + getAddress();
   }
   void join(Organization org) {
      org.addMember(this);
   }
}

class Organization {
   Set<Person> members = new HashSet<Person>();
   void addMember(Person p) { members.add(p); }
   boolean hasMember(Person p) {
      return members.contains(p);
   }
}

/*role*/ class Employee extends Person {
   Employer employer;
   Employee(String n, String a, Employer e) {
      super(n, a);
      employer = e;
   }
   @Override String getAddress() {
      return employer.getAddress();
   }
}

class Employer {
   String address;
   Employer(String a) { address = a; }
   String getAddress() { return address; }
}

/*role*/ class Agent extends Person {
   Agent(String n, String a) { super(n, a); }
}

class Main {
   static void main(String... args) {
      Employer gvnmt = new Employer("London");
      Employee bond =
         new Employee("Bond", "secret", gvnmt);
      assert bond.getContactInfo()
         .equals("Bond, London") : "Assertion 1";
      Agent double07 = new Agent("Bond", "secret");
      Organization mi6 = new Organization();
      double07.join(mi6);
      assert mi6.hasMember(double07) : "Assertn 2";
      assert mi6.hasMember(bond) : "Assertion 3";
   }
}
```

Limitation of Sub-Classes based implementation of Roles

- From an object identity point of view, subclassing implies that two objects of different subclasses are not identical, they are different.
- Thus, a Person acting both as Agent and Employee is represented by two different objects with distinct identities.
- Here, both "bond" and "double07" are meant to represent the same entity "person", they are actually different. This is evidenced by the failure of Assertion 3

Solution Two

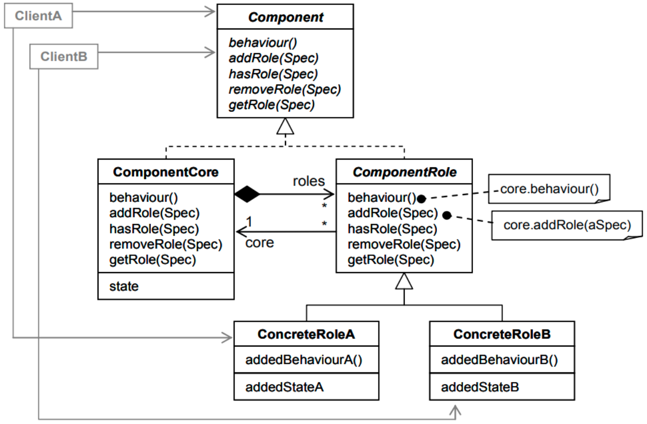
Role Object Pattern

- The Role Object pattern suggests to model context-specific views of an object as separate role objects which are dynamically attached to and removed from the core object.
- The resulting composite object structure is consisting of the Component, Componentcore, ComponentRoles and ConcreteRole .
- Component: It specifies the protocol for adding, removing, testing and querying for role objects.

Role Object Pattern Cont.

- ComponentCore: It implements the Component interface including the role management protocol. It also creates ConcreteRole instances, manages its role objects.
- ComponentRole: It stores a reference to the ComponentCore, implements the Component interface by forwarding requests to its core attribute
- ConcreteRole: It implements a context-specific extension of the Component interface.It also can be instantiated with a ComponentCore as argument

Structure of ROP



Implementation in ROP

```java
interface Person {
   String getAddress();
   String getContactInfo();
   void join(Organization org);
   <R extends PersonRole> R addRole
        (Class<R> spec, Object... arguments);
   ...
}

class PersonCore implements Person {
   private String name, address;
   private Collection<PersonRole> roles =
        new HashSet<PersonRole>();
   PersonCore(String n, String a) {
      name = n; address = a;
   }
   String getAddress() { return address; }
   String getContactInfo() {
      return name + ", " + getAddress();
   }
   void join(Organization org) {
      org.addMember(this);
   }
   <R extends PersonRole> R addRole
           (Class<R> spec, Object... arguments) {
      R role = ...
      roles.add(role); role.core = this;
      return role;
   }
   ...
}
```

```java
abstract class PersonRole implements Person {
   PersonCore core;
   String getAddress() {
      return core.getAddress();
   }
   String getContactInfo() {
      return core.getContactInfo();
   }
   void join(Organization org) {
      core.join(org);
   }
   <R extends PersonRole> R addRole
           (Class<R> spec, Object... arguments)
      return core.addRole(spec, arguments);
   }
   ...
}

class Employee extends PersonRole {
   ...
   Employee(Employer e) { employer = e; }
}
```

```java
class Agent extends PersonRole {
   Agent() {}
}

class Main {
   static void main(String... args) {
      Person james =
           new PersonCore("Bond", "secret");
      Employer gvnmt = new Employer("London");
      Employee bond =
           james.addRole(Employee.class, gvnmt);
      assert bond.getContactInfo()
           .equals("Bond, London") : "Assertion 1";
      Agent double07 = james.addRole(Agent.class);
      Organization mi6 = new Organization();
      double07.join(mi6);
      assert mi6.hasMember(double07) : "Assertn 2";
      assert mi6.hasMember(bond) : "Assertion 3";
   }
}
```

Implementation in ROP Cont.

- The following code is the naive implementation of Riehle's Role Object pattern
- The main difference between the original design and the ROP is that the original Person class is now an interface serving as a common abstraction of a new PersonCore class and a new abstract PersonRole class.
- The two new classes PersonCore and PersonRole both form a composition in the sense that a core and its role objects together represent one logical entity
- All roles which must be subclasses of PersonRole, can add their own state and behaviour, complementing those of PersonCore.

Problem of Role Object Pattern

- The following code is expected to hold Assertions 3 now, instead all three Assertion failed.
- Assertion one failed because ROP introduced changed of binding. Instead of returning the address of the employer as specified in class Employee, it returns the private address from class PersonCore .
- The problem is that PersonRole.getContactInfo() uses forwarding where it should have used delegation.
- With delegation the delegated method (PersonCore.getContactInfo()) would refer to the instance of the delegator (here: bond, an instance of class Employee) rather than to an instance of the delegatee (here: james, instance of PersonCore)

Refactoring naive ROP implementation

- To introduce Delegation to our problem, we need some refactor in our code. We need to add a parameter to the delegated method that refers back to the delegator.
- Here is the following change which will restore Assertion 1

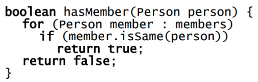
```
public String getContactInfo() {
    return getContactInfo(this);
}
public String getContactInfo(Person del) {
    return name + ", " + del.getAddress();
}
```

Refactoring naive ROP implementation Cont.

- Assertion 2 still fails because, when the invocation of join(Organization) on doubleO7 is forwarded to PersonCore, the object this that is to join the organization changes from the delegator (doubleO7, an instance of Agent) to the delegatee (james, an instance of PersonCore).
- Thus, when the checked for the membership of the newly joined Agent instance "doubleO7", the answer is no, since it was the PersonCore instance that joined.
- The problem can be fixed by passing "doubleO7" as an additional delegator parameter to the PersonCore.join() method and by replacing "this" in its body with delegator

Refactoring naive ROP implementation Cont.

- Assertion 3 still remains as a failure which is caused by object schizophrenia. In ROP, a role is splited into two instances
- So, tests for identity of two references depend on whether both references together either point to the role or the core instance. I
- If identity of the logical subject which, depending on context, may be represented by the core or any of its roles, is to be tested, a test for identity is needed.



```
boolean hasMember(Person person) {
  for (Person member : members)
    if (member.isSame(person))
      return true;
  return false;
}
```

Drawbacks of Refactoring of naive ROP impl.

- Popular, implementation of the ROP in programming languages without delegation as a native language construct suffers from a number of non-negligible problems.
- Firstly, to emulate the delegation requires auxiliary methods that bloat class interfaces and implementation and, because of the additional method dispatches introduced, lead to code that is difficult to understand and maintain.
- Secondly, the object schizophrenia resulting from splitting an object into a role and a core requires the introduction of delegation parameters even in cases in which method binding can remain unchanged

Drawbacks of Refactoring of naive ROP impl. Cont.

- Another issue with introducing the ROP is that the necessary refactoring of the program inherits some rather strong preconditions from its sub-refactorings, in particular "EXTRACT INTERFACE" and "REPLACE INHERITANCE WITH DELEGATION"
- While solving an important practical problem it seems that adopting the ROP is subject to so many "ifs", "ands", and "buts" that the search for a better implementation of role objects is justified.

Refactoring Towards LOR

Lightweight Role Object Pattern

- In object-oriented programming languages delegation can be also implemented as inheritance, and that all the extra methods and parameters required by explicit delegation are made obsolete by applying the "REPLACE DELEGATION WITH INHERITANCE" refactoring
- The problem of replacing the delegation of the ROP with inheritance is that, it deprives two objects sharing behaviour of their shared state. Becasue in class-based object oriented programming languages, every object receives its own copy of the fields inherited from its super classes.

Lightweight Role Object Pattern Cont.

- To overcome the sharing problem, we can make the shared state to an externalized object to which all objects involved in the sharing can refer
- This object must be an instance of a new class introduced for this purpose, and in particular that it cannot be an instance of the entity class
- Sharing state in this manner is similar to the delegation required for the ROP in that a single entity and its role objects share an object holding the common state
- here, the shared object does not represent the entity itself instead, the entity has its state externalized in exactly the same manner as the role objects.

Refactoring towards LRO

The ROP can thus be refactored as follows:

- By applying the "EXTRACT CLASS" refactoring to the core class, move all fields of the core class to a new state class, and assign the core class a single new field state of the type of the state class, which holds an instance keeping the core class's instances' externalized state.
- Using the REPLACE DELEGATION WITH INHERITANCE refactoring, replace the delegation from the (abstract) role class to the core class by inheritance, making the former a subclass of the latter. The delegation (sharing) of state is achieved by assigning the "state" field inherited from the core class
- The former core class is now a superclass of the role classes, the common interface of the core and the roles has become obsolete and can be removed

Figure of LRO

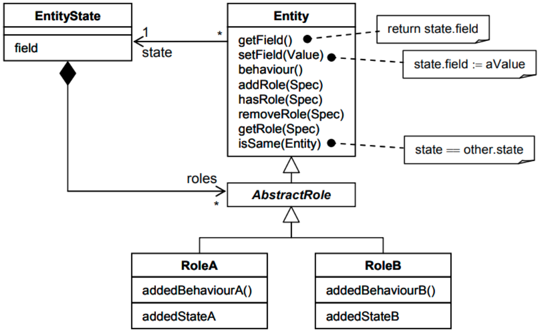
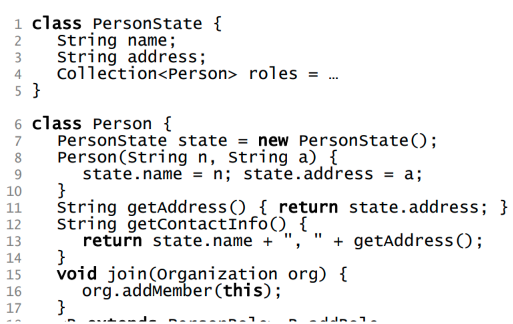
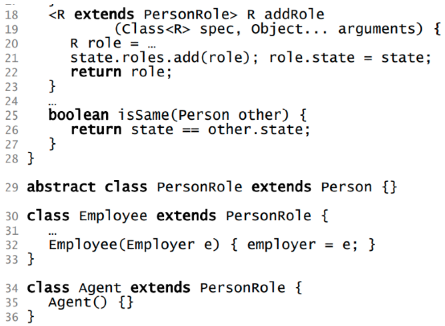


Figure 6: Lightweight role objects: roles as subclasses of an Entity class with externalized state and identity.

Sample Implementation of LRO



```java
class PersonState {
   String name;
   String address;
   Collection<Person> roles = …
}
class Person {
   PersonState state = new PersonState();
   Person(String n, String a) {
      state.name = n; state.address = a;
   }
   String getAddress() { return state.address; }
   String getContactInfo() {
      return state.name + ", " + getAddress();
   }
   void join(Organization org) {
      org.addMember(this);
   }
```



```java
   <R extends PersonRole> R addRole
            (Class<R> spec, Object... arguments) {
      R role = …
      state.roles.add(role); role.state = state;
      return role;
   }
   …
   boolean isSame(Person other) {
      return state == other.state;
   }
}
abstract class PersonRole extends Person {}
class Employee extends PersonRole {
   …
   Employee(Employer e) { employer = e; }
}
class Agent extends PersonRole {
   Agent() {}
}
```

Conclusion

Conclusion

- While the Role Object pattern describes a widely accepted solution for the problem of representing objects with roles in class-based object-oriented systems, we have found that its technical requirements, especially concerning the necessary delegation, seriously question its practical utility.
- By separating shared state and identity from the sharing of behaviour, we have managed to preserve inheritance as the native delegation mechanism required by role objects, allowing us to introduce a lightweight form of role objects.

Thank You!