3F6 - Software Engineering and Design

Handout 5

Object Oriented Design

With Markup

Ed Rosten

Contents

Copies of these notes plus additional materials relating to this course can be found at: `http://mi.eng.cam.ac.uk/~er258/teaching`.

Object-oriented design

Key Ideas:

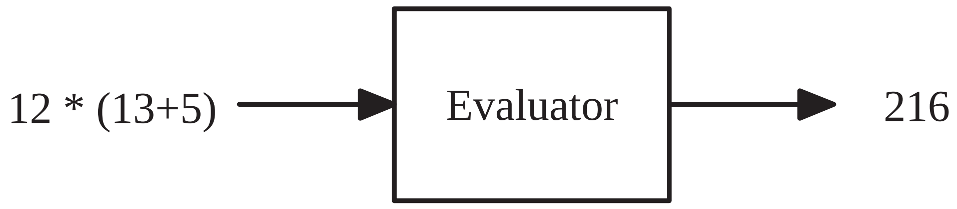
- **Encapsulation:**
  use classes to hide implementation details and provide well-defined interfaces.
- **Composition:**
  use objects as building bricks to construct complex systems from more manageable sub-components.
  Focus on **has-a** relationship.
- **Inheritance:**
  use class inheritance to extend and re-use existing objects.
  Focus on **is-a** relationship.
- **Polymorphism:**
  use virtual functions to allow related (but not identical) objects to be treated as a homogeneous group.

Guiding principles

- Ensure a clear specification.
- Model the core data structures first.
- Group the required functionalities into logically distinct modules.
- Design with re-use and extensibility in mind.
- Every class should be responsible for itself.
- Make class interfaces as secure as possible.
- Design for error handling from the outset.

An arithmetic expression evaluator

Consider the design of an evaluator for simple arithmetic expressions:



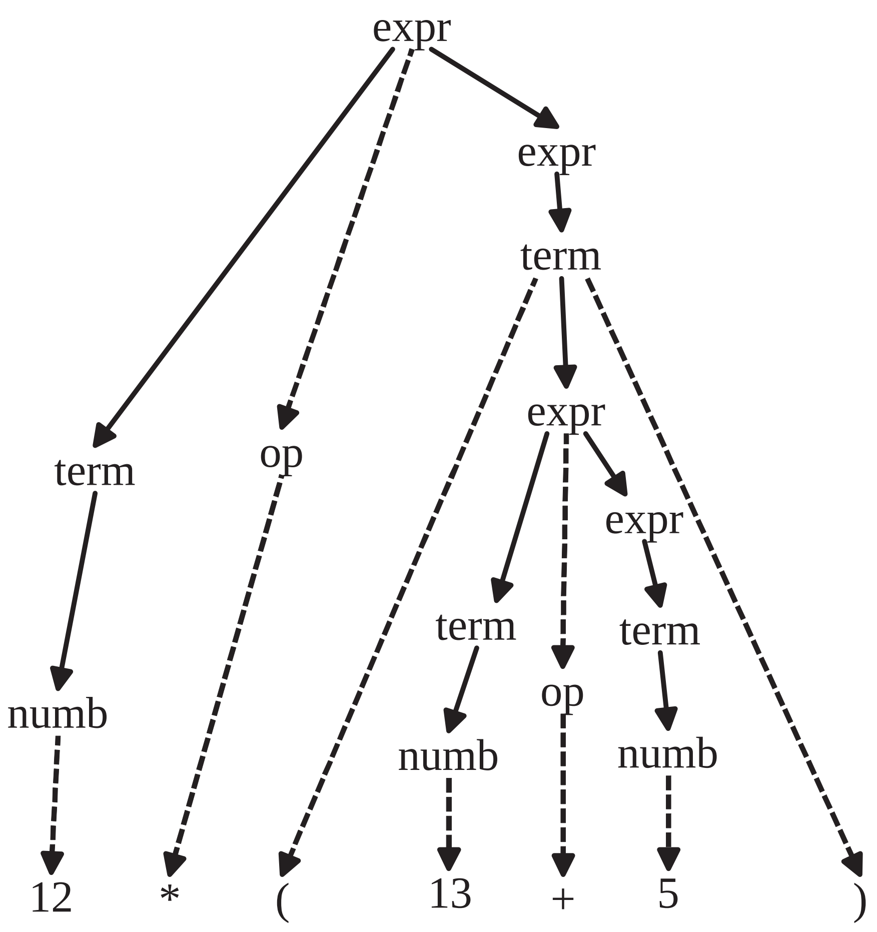
Step One Refine the specfication:

1. What constitutes a legal expression?
   This is specified by the syntax rules shown below. All operators have equal precedence and no specific evaluation order. A typical application of these rules is shown also in the form of a *parse tree*.
2. Are there any design constraints?
   Yes, the design should include an explicit internal representation of the expression in memory to allow additional functions to be added (eg to allow the structure of expressions to be displayed).
3. What should happen on illegal inputs?
   An error message should be displayed which indicates the type of error and the point in the expression at which it occurs.

Expression Syntax

```
expr => term   [   op  expr   ]
term => numb  |  "(" expr ")" | "-" term
op     => "+" | "-" | "*" | "/"
numb   =>  digit [ numb ]
```

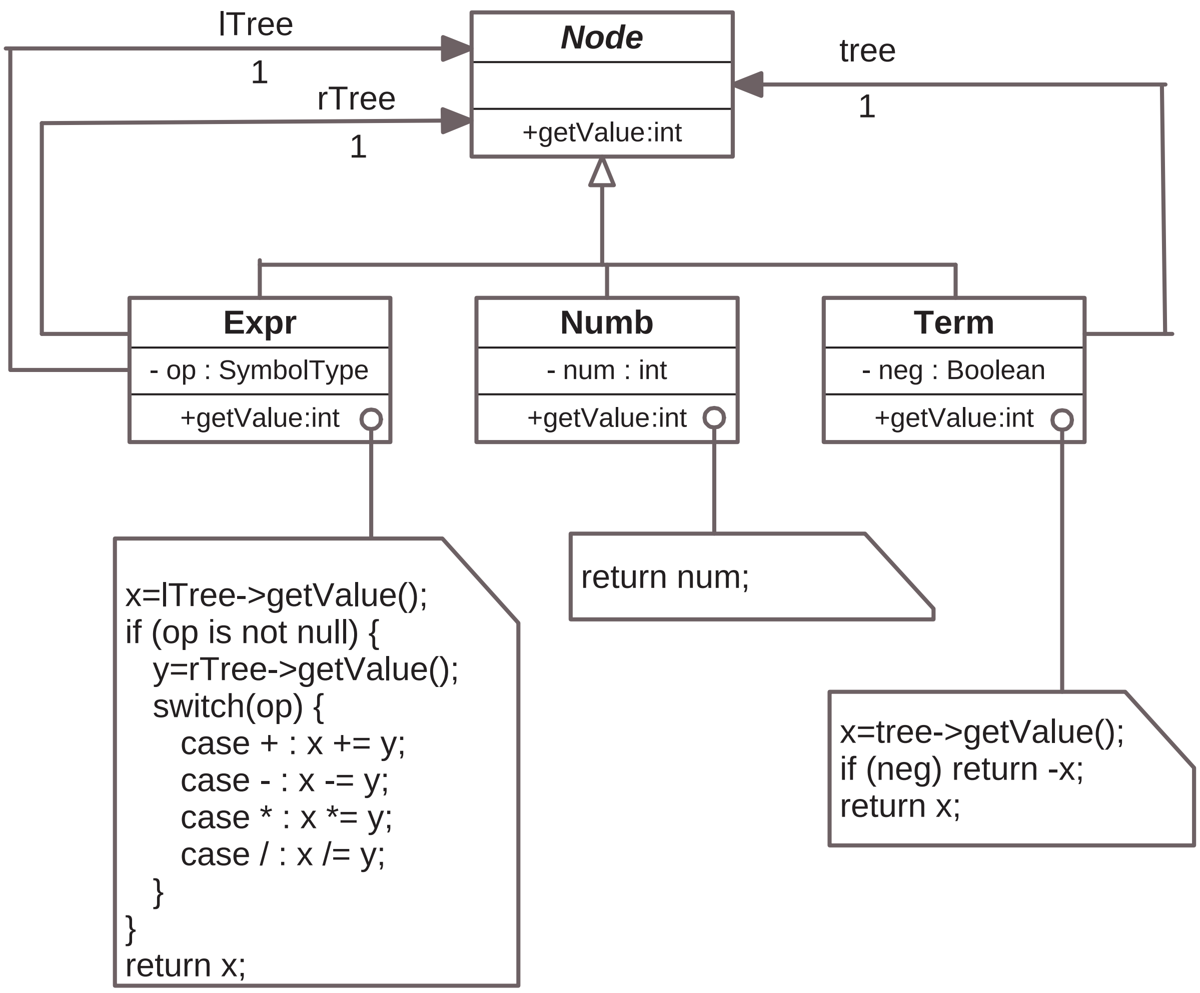
Parse tree of a typical expression



[NB solid arrows translate into pointers to objects in the following design]

Step Two Design the core data structures:

To support the display of expression structure, associate each node of the parse tree with an object. To support polymorphic processing of tree nodes, define an abstract class `Node` and then derive classes for `Expr`essions, `Term`s, and `Numb`ers from it. Note that a class for operators is not needed since they can always be stored with the parent expression.



```
x=lTree->getValue();
if (op is not null) {
   y=rTree->getValue();
   switch(op) {
      case + : x += y;
      case - : x -= y;
      case * : x *= y;
      case / : x /= y;
   }
}
return x;
```

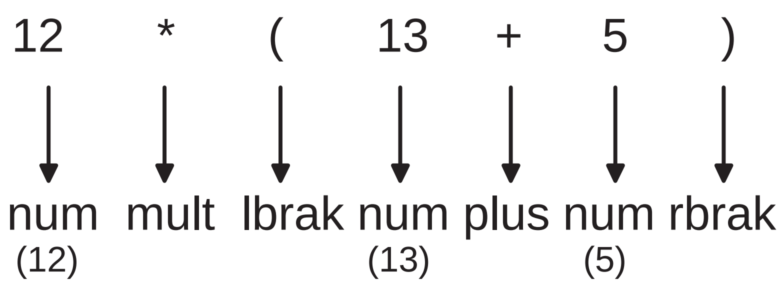
```
return num;
```

```
x=tree->getValue();
if (neg) return -x;
return x;
```

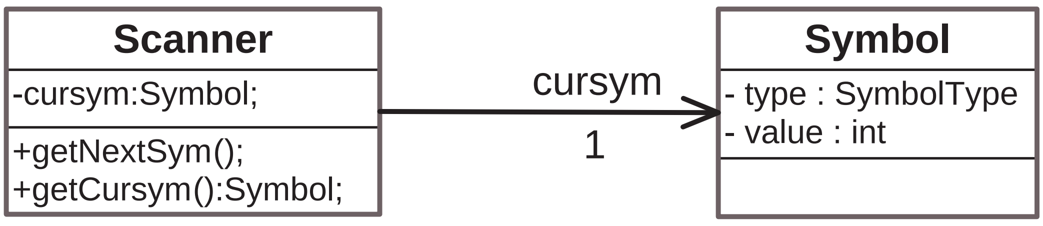
Implementation of `getValue()` illustrates the "Every class should be responsible for itself" principle.

Step Three Define the auxiliary interfaces:

Here the main issue is the design of the input processing. The conventional approach is to assume the existence of a *scanner* which reads the input character sequence and converts it to a sequence of logical symbols:



The scanner maintains a *current symbol* and the interface consists of calls to read the current symbol and get the next symbol:



where the `SymbolType` is a simple enumeration of all the possible symbol types eg

```
    enum SymbolType {num,lbrak,rbrak,plus,minus ....}
```

Step Four Construction of the core data structures

Exploit the "Every class should be responsible for itself" principle again:

```
// expr => term [ op expr ]
Expr::Expr(Scanner * s)
{
   lTree = new Term(s);  rTree = NULL;
   if (s->getCurSym() is operator) {
      op = s->getCurSym().Op()  // store the actual op
      s->getNextSym();
      rTree = new Expr(s);
   }
}

// term => "(" expr ")" | "-" term | number
Term::Term(Scanner * s)
{
   neg = false; tree = NULL;
   if (s->getCurSym() is lbrak) {
      s->GetNextSym();
      tree = new Expr(s);
      // check current sym is rbrak
      s->GetNextSym();
   }else if (s->getCurSym() is minus) {
      s->GetNextSym(); neg = true;
      tree = new Term(s);
   }else {
      // check current sym is number
      tree = new Numb(s);
   }
}
```

Note that a scanner object is passed to each constructor.

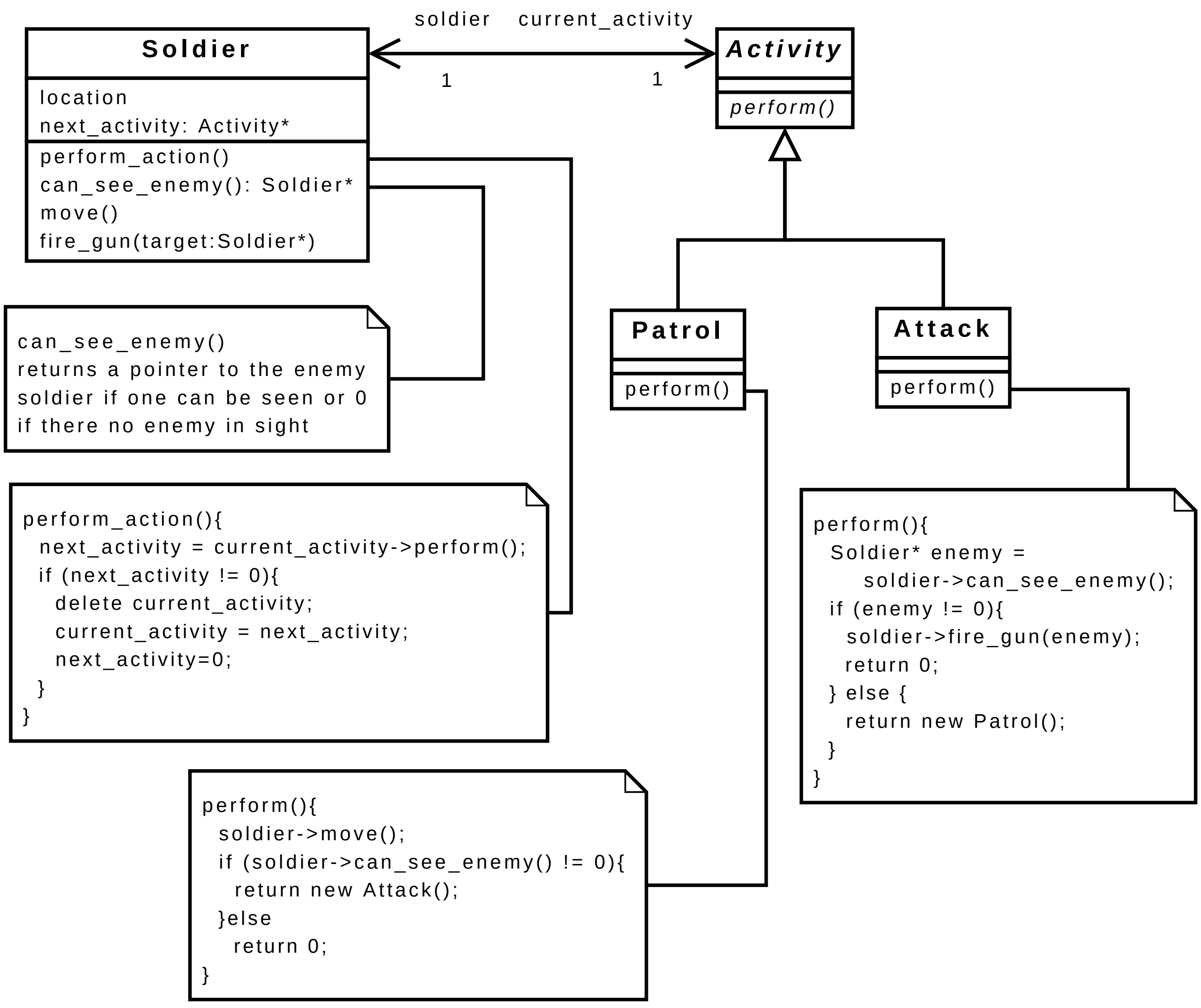
The basic design of the expression parser is now complete. All that remains is some implementation detail. A full working program can be inspected at
`http://mi.eng.cam.ac.uk/~er258/teaching`.

Some points to note about this design.

- Given a very precise definition of the inputs via formal grammar rules and their corresponding parse trees, it is straightforward to map syntax nodes to derived classes. Each derived class is responsible for parsing its own syntax rule.
- Polymorphism simplifies the processing of syntax nodes.
- The object oriented design makes the addition of extra functionality straightforward. Consider the requirement to print out the structure - this is easily achieved by adding simple print routines to each node type.
- The definition of the input scanner as a class object makes it easy to extend to other input sources (eg a file) without changing the rest of the program.
- Separation of the scanner from the parser via a secure interface makes it easy to delegate implementation of the scanner and parser to different members of a team.

War Game Simulator

A company makes and sells computer war games. The figure below shows a UML class diagram which describes part of the software which is responsible for the artificial intelligence of computer controlled soldiers in the game.



```
can_see_enemy()
returns a pointer to the enemy
soldier if one can be seen or 0
if there no enemy in sight
```

```
perform_action(){
  next_activity = current_activity->perform();
  if (next_activity != 0){
    delete current_activity;
    current_activity = next_activity;
    next_activity=0;
  }
}
```

```
perform(){
  Soldier* enemy =
      soldier->can_see_enemy();
  if (enemy != 0){
    soldier->fire_gun(enemy);
    return 0;
  } else {
    return new Patrol();
  }
}
```

```
perform(){
  soldier->move();
  if (soldier->can_see_enemy() != 0){
    return new Attack();
  }else
    return 0;
}
```

What does this diagram say?

There are four classes shown in the diagram: Soldier, Activity, Patrol and Attack.

Each Soldier has an activity called current activity and each activity has a soldier (i.e. a 1:1 relationship). Patrol and Attack are derived from (or subclasses of) Activity. Activity is abstract and hence each activity is either a Patrol or an Attack.

The separation of Soldiers from the Activities that they perform allows both to be refined and extended independently. This is a familiar design pattern called the *state* pattern which we will see more of later.

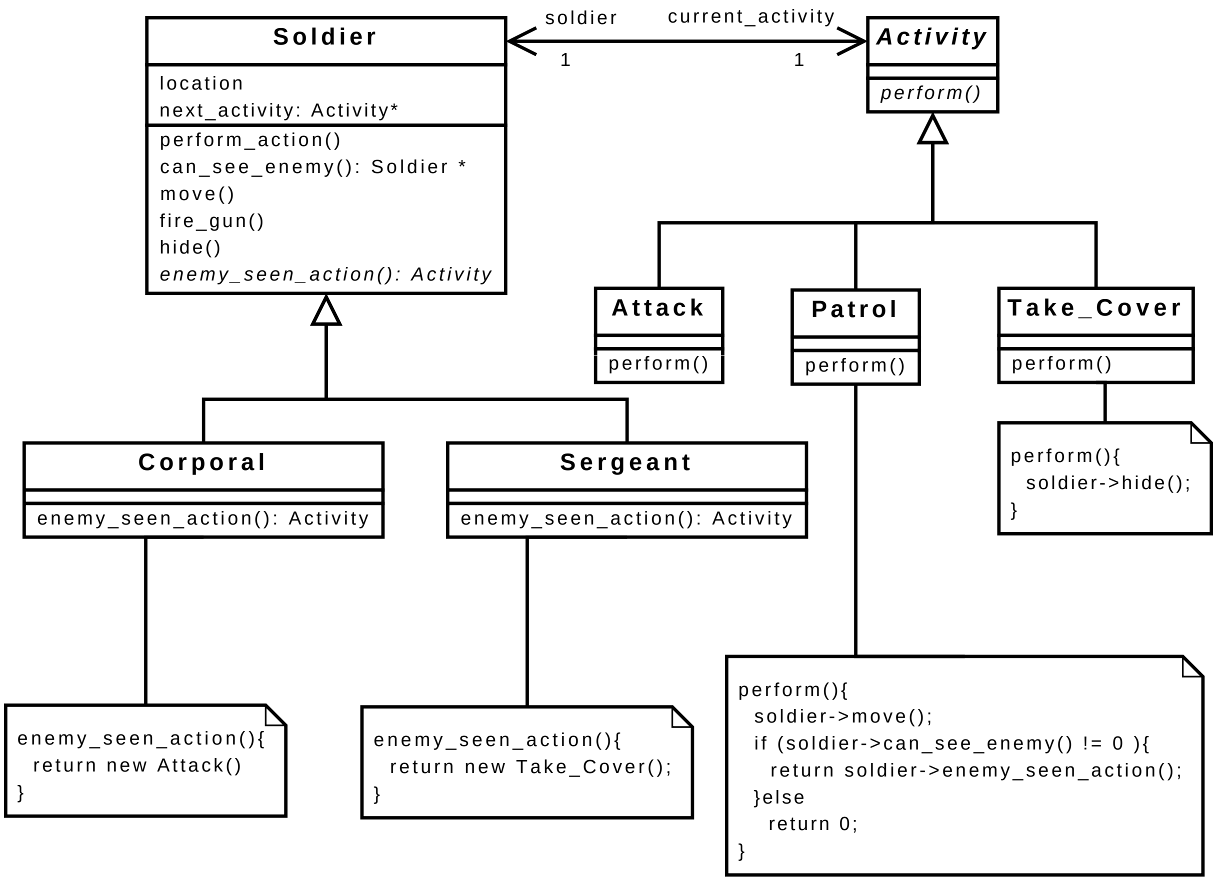
Additional Functionality

It is now decided that there will be two kinds of soldier within the game: Corporal and Sergeant. A Corporal will continue to follow the existing behaviour of a Soldier in the current game. A Sergeant's behaviour will differ so that when they are on patrol and see the enemy, a Sergeant will engage in a new kind of activity: "Take_Cover".

Assuming that an additional function called hide() which implements the act of taking cover has been added to the Soldier class, what software changes are required to support this new behaviour?

Solution

1. Add a new class `Take_Cover` derived from `Activity` just like `Patrol` and `Attack`.
2. There are now two kinds of soldier. This implies that the `Soldier` class needs to be replaced by a class hierarchy in which `Corporal` and `Sergeant` are derived from `Soldier`.
3. There should also be a virtual function in this hierarchy to embody the differing behaviours. This differing behaviour is the state that the soldier should move into when an enemy is seen thus the virtual function should return this new state.



```
perform(){
  soldier->hide();
}
```

```
enemy_seen_action(){
  return new Attack()
}
```

```
enemy_seen_action(){
  return new Take_Cover();
}
```

```
perform(){
  soldier->move();
  if (soldier->can_see_enemy() != 0 ){
    return soldier->enemy_seen_action();
  }else
    return 0;
}
```

Sequence Diagram

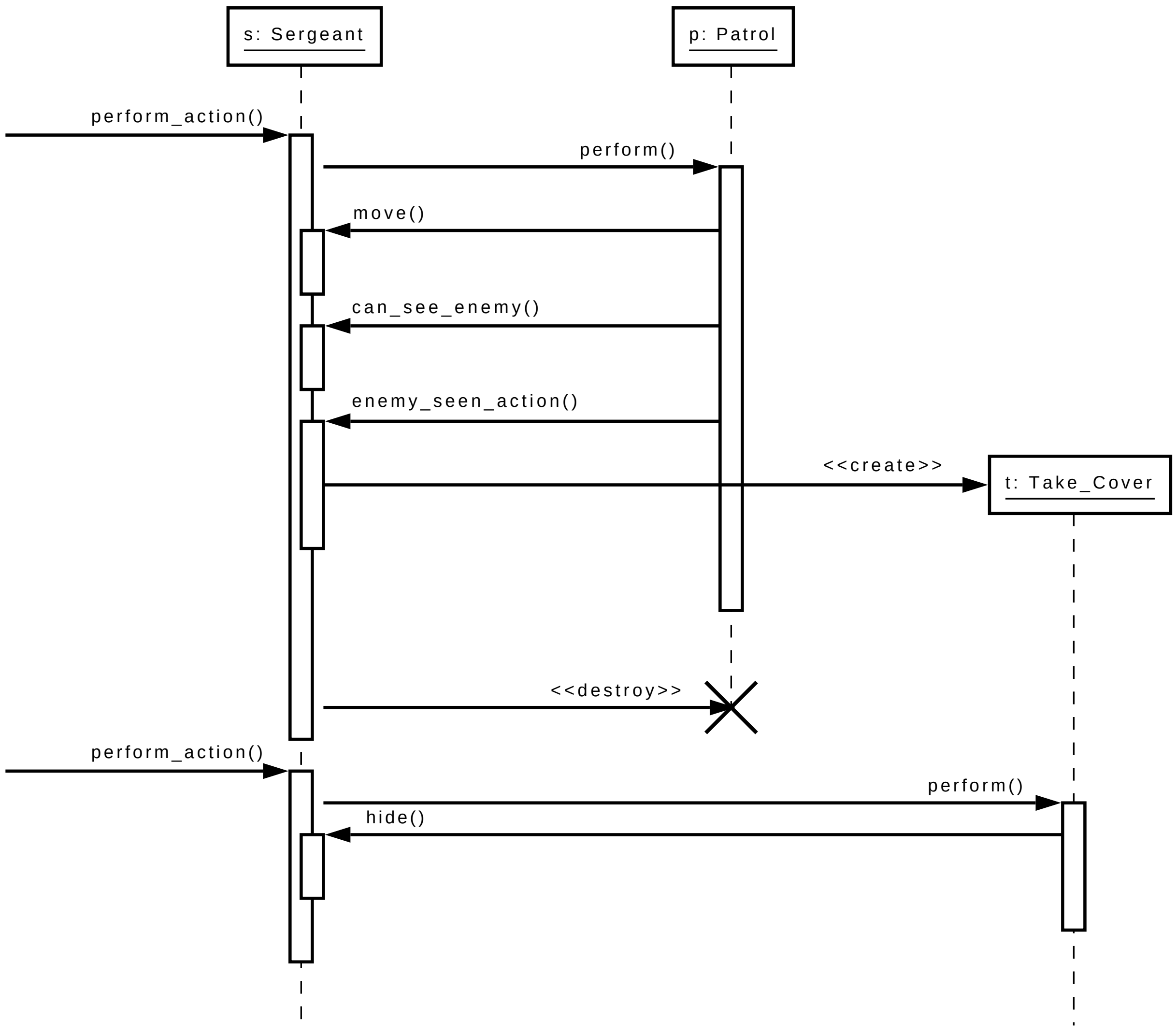
Show what happens when `perform_action()` is called for a patrolling sergeant and he sees the enemy.

Solution

The diagram must show the following sequence of events:

1. `perform_action()` is called on the sergeant.
2. The sergeant calls `perform()` on its activity which is a `Patrol` object.
3. The `Patrol` object calls `move()` on the soldier;
4. The `Patrol` object calls `can_see_enemy()` on the sergeant which will return a pointer to the soldier that can be seen.
5. This causes the `Patrol` object to call the virtual `enemy_seen_action()` function on the sergeant.
6. The sergeant creates a new `Take_Cover` object and returns it to the `Patrol` object.
7. The `Patrol` object then returns the `Take_Cover` object back to the Sergeant.
8. The Sergeant now completes the `perform_action()` function by destroying the original `Patrol` object and storing a reference to the new `Take_Cover` object in its place so that any subsequent call to `perform_action()` will call the `perform()` function in the new `Take_Cover` object (see the

note containing the **`perform_action()`** code in the UML diagram).



A Further Extension

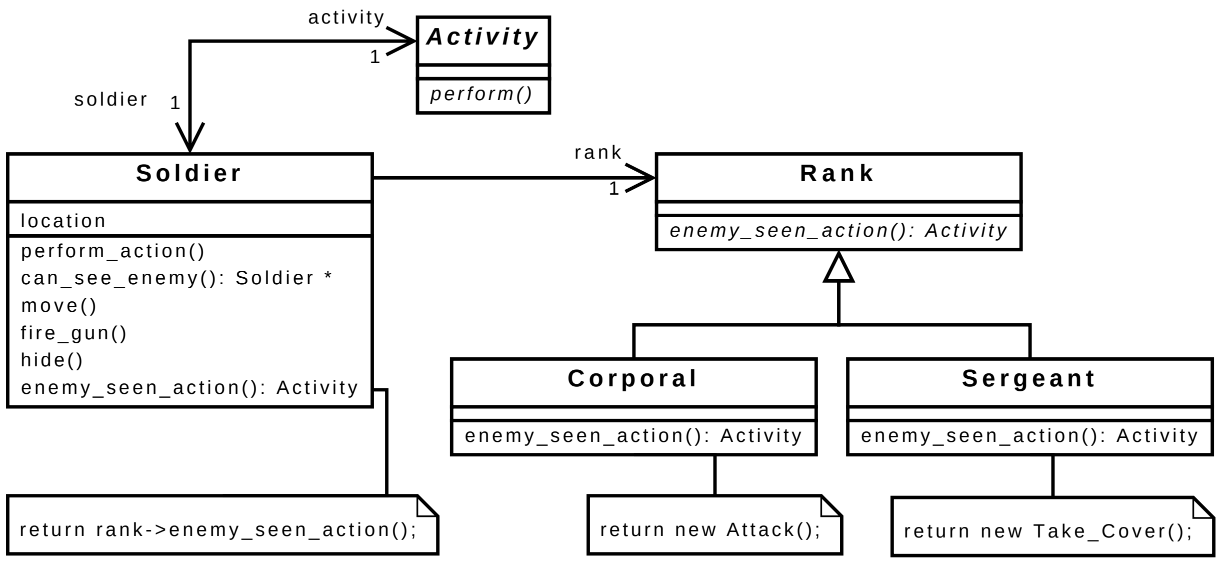
Suppose that it is now possible for a Corporal to be promoted to Sergeant during the game. How must the software be updated?

Solution

If a soldier can be promoted, this means that `Corporal` and `Sergeant` can not be derived from `Soldier` since this would prevent a soldier changing rank. The solution is to make a soldier have a rank object which can change. This means that a new `Rank` class is needed which is the superclass for the `Corporal` and `Sergeant` classes which house the virtual function.

The `enemy_seen_action()` function in `Soldier` must *forward* the call to the `enemy_seen_action()` function to the `Rank` object.

The subclasses of `Activity` have been omitted from the following diagram since they remain unchanged.


activity
1
Activity
perform()
soldier
1
Soldier
location
perform_action()
can_see_enemy(): Soldier *
move()
fire_gun()
hide()
enemy_seen_action(): Activity
rank
1
Rank
enemy_seen_action(): Activity
Corporal
enemy_seen_action(): Activity
Sergeant
enemy_seen_action(): Activity
return rank->enemy_seen_action();
return new Attack();
return new Take_Cover();