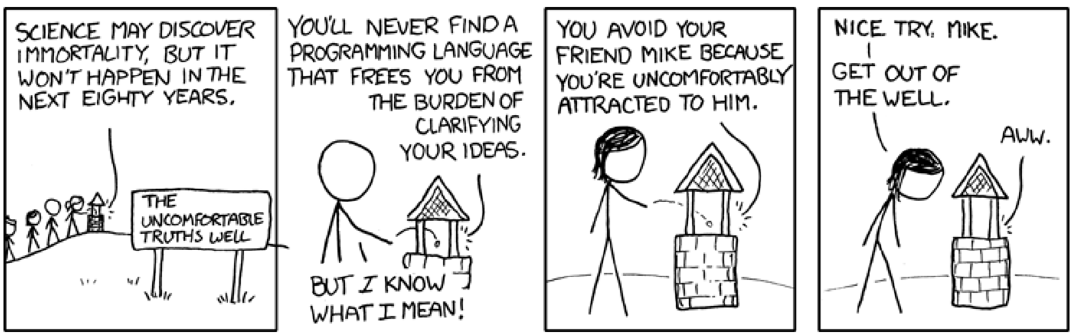
Assignment 7: Inheritance and polymorphism

ETH Zurich

Hand-out: Monday, 5 November 2012
Due: Wednesday, 14 November 2012



Well 2 © Randall Munroe (http://xkcd.com/568/)

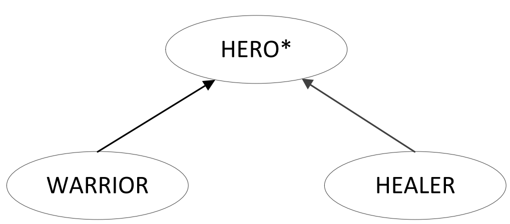
Goals

- Understand polymorphism and dynamic binding.
- Practice inheritance.
- Continue the design and implementation of the board game.

1 Polymorphism and dynamic binding

Review polymorphic attachment and dynamic binding (Touch of Class, sections 16.2 and 16.3).

Below you can see a class diagram and code of three classes from a new video game "Blades of Glory".



```
deferred class
  HERO

feature -- Initialization
  make (s: STRING)
      -- Create a hero with name 's'.
    require
      s /= Void
    do
      name := s
      level := 1
      health := 100
    end

feature -- Access
  name: STRING

  level: INTEGER

  health: INTEGER

feature -- Basic operations
  do_action (other: HERO)
      -- Perform main action on 'other'.
    require
      alive: health > 0
    deferred
    end

  level_up
      -- Increase level.
    do
      level := level + 1
      set_health (100)
    end

feature {HERO} -- Setters
  set_health (h: INTEGER)
      -- Set 'health' to 'h'.
    require
      0 <= h and h <= 100
    do
      health := h
      if health = 0 then
        print (name + " is dead.%N")
      end
    end

invariant
  name /= Void
  0 <= health and health <= 100
  level > 0
```

```
end

class
  WARRIOR

inherit
  HERO
    rename
      do_action as attack
    redefine
      level_up
    end

create
  make

feature -- Basic operations
  attack (other: HERO)
      -- Attack 'other'.
    local
      damage: INTEGER
    do
      damage := (5 * level).min (other.health)
      other.set_health(other.health - damage)
      print (name + " attacks " + other.name + ". Does " + damage.out + " damage%
          N")
    end

  level_up
    do
      Precursor
      print (name + " is now a level " + level.out + " warrior%N")
    end

end

class
  HEALER

inherit
  HERO
    rename
      do_action as heal
    redefine
      make,
      level_up
    end

create
  make

feature -- Initialization
```

```
  make (s: STRING)
      -- Create a healer with name ‘s’.
    do
      Precursor (s)
      mana := 100
    end

feature -- Access
  mana: INTEGER

feature -- Basic operations
  heal (other: HERO)
      -- Heal ‘other’.
    local
      h: INTEGER
    do
      if mana >= 10 then
        h := (10 * level).min (100 - other.health)
        other.set_health(other.health + h)
        mana := mana - 10
        print (name + " heals " + other.name + " by " + h.out + " points%N")
      end
    end

  level_up
    do
      Precursor
      mana := 100
      print (name + " is now a level " + level.out + " healer%N")
    end
end
```

Given the following variable declarations:

```
hero: HERO
warrior: WARRIOR
healer: HEALER
```

indicate, for each of the code fragments below, whether it compiles. If the code fragment does not compile, explain why this is the case. If the code fragment compiles, specify the text that is printed to the screen when the code fragment is executed. This is a pen-and-paper task; you are not supposed to use EiffelStudio.

Example:

```
create warrior
warrior.level_up
```

This code does not compile, because default creation is not available for class *WARRIOR*.

Task 1

```
create warrior.make ("Thor")
warrior.level_up
```

Task 2

```
create hero.make ("Althea")
hero.level_up
```

Task 3

```
create warrior.make ("Thor")
create healer.make ("Althea")
warrior.do_action (healer)
```

Task 4

```
create {HEALER} warrior.make ("Diana")
warrior.level_up
```

Task 5

```
create {WARRIOR} hero.make ("Thor")
hero.do_action (hero)
create {HEALER} hero.make ("Althea")
hero.do_action (hero)
```

Task 6

```
create {WARRIOR} hero.make ("Thor")
warrior := hero
warrior.attack (hero)
```

To hand in

Hand in your answers for the code fragments above.

2 Ghosts in Zurich

Ghosts are taking over Zurich! In this task you will implement a special kind of mobile object: a *GHOST*. Ghosts in Traffic have the following behavior: they choose a station of the city and then fly around it in circles.

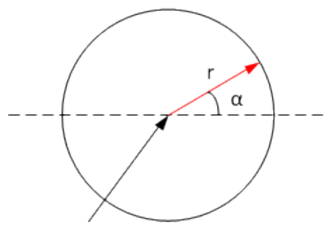
To do

1. Download `http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/07/traffic.zip`, unzip it and open `assignment_7.ecf`.
2. Create a new class *GHOST* and make it inherit from *MOBILE*. The latter has three deferred features: *position*, *speed* and *move_distance*, which you have to implement before you can successfully compile your class. For the first two features you have a choice of making them into either an attribute or a function. The third one should be implemented as a procedure that calculates where the ghost ends up when it moves from the current

position by d meters. You can assume that all ghosts always move at the same speed (e.g. 10 meters per second).

You'll probably also want to add new features to *GHOST*, for example to store the station that it is flying around and the distance it keeps from the station (the radius of its circular trajectory). Additionally you'll need a creation procedure that takes the station and the radius as arguments.

Hint: It's convenient to represent the ghost position at any point in time as a sum of two vectors, one of them constant and the other one changing as the ghost moves, like on this picture:



3. In the class *GHOST_INVASION* implement a feature *add_ghost* (*s*: *STATION*; *r*: *REAL_64*) that creates a ghost flying around a station *s* at a distance *r* and adds it to Zurich (using the feature *add_custom_mobile*). Don't forget to update the map in order to create the view for the new ghost. After that, modify the view so that the ghost is depicted as an icon instead of the default black dot; you can use "ghost.png" from the "images" directory for the icon. The expression *Zurich_map.custom_mobile_view* (*ghost*) gives you access to the view of the object *ghost*.

   Test the *add_ghost* feature by calling it from *invade* with arguments of your choice. To make the ghost move, double-click on the map.

4. Modify the feature *invade* so that it generates 10 ghosts flying around random stations of Zurich at a random distance between 10 and 100 meters (you don't have to check that all stations are different). To access stations by integer index, create a cursor that iterates through the stations and call the command *go_to* on that cursor.

To hand in

Hand in classes *GHOST* and *GHOST_INVASION*.

3 Code review

Code review is a widely applied software engineering practice, in which source code produced by a software developer is examined by his or her peers. The purpose of a code review is to find design, programming, and style errors, improving the overall software quality and the developers' skills.

In this task you will conduct a review of the Board game (part 2) implementation, written by one of your peers. You will receive the code to review from your assistant by the end of Wednesday, November 7.

To do

Examine the code carefully, evaluating the following aspects:

1. Choice of abstractions (the set of classes, the set of responsibilities of each class)
2. Architecture (relationship between classes, such as inheritance and client-supplier)
3. Contracts
4. Implementation techniques (choice of data structures and algorithms)
5. Coding style and names
6. Comments and documentation (including header comments and **note** clauses for classes)

For each category listed above, write down related issues you found in the code, if any. If the same issue occurs multiple times (for example, a header comment is missing in *all* features) you only have to mention it once.

To hand in

Your review.

4 Board game: Part 3

In this task you will extend the implementation of the board game. You will find an updated problem description below.

The board game comes with a board, divided into 40 squares, a pair of six-sided dice, and can accommodate 2 to 6 players. It works as follows:

- All players start from the first square.
- One at a time, players take a turn: roll the dice and advance their respective tokens on the board.
- A round consists of all players taking their turns once.
- Players have *money*. Each player starts with 7 CHF.
- The amount of money changes when a player lands on a special square:
  - Squares 5, 15, 25, 35 are *bad investment* squares: a player has to pay 5 CHF. If the player cannot afford it, he gives away all his money.
  - Squares 10, 20, 30, 40 are *lottery win* squares: a player gets 10 CHF.
- The winner is the player with the most money after the first player advances beyond the 40th square. Ties (multiple winners) are possible.

To do

Modify the implementation of the board game in such a way that it accommodates the changes in the problem description (money, special squares, new winning criterion). We recommend that you start from the master solution to the assignment 6: `http://se.inf.ethz.ch/courses/2012b_fall/eprog/assignments/07/board_game.zip`.

Hints

Are there entities in the problem domain that didn't have enough properties and behavior to deserve their own classes in the previous version of the game, but that gained some properties or behavior in the current version? You might want to introduce new classes for such entities.

Bad investment and lottery win squares are special cases of squares, which differ in a way they affect players. To model this you can introduce class *SQUARE* and then use inheritance and feature redefinition to implement the behavior of special squares. You can store squares of all kinds in a single polymorphic container (e.g. *V_ARRAY* [*SQUARE*]) and let dynamic binding take care of which special behavior applies for each square.

To hand in

Hand in the code of your classes.