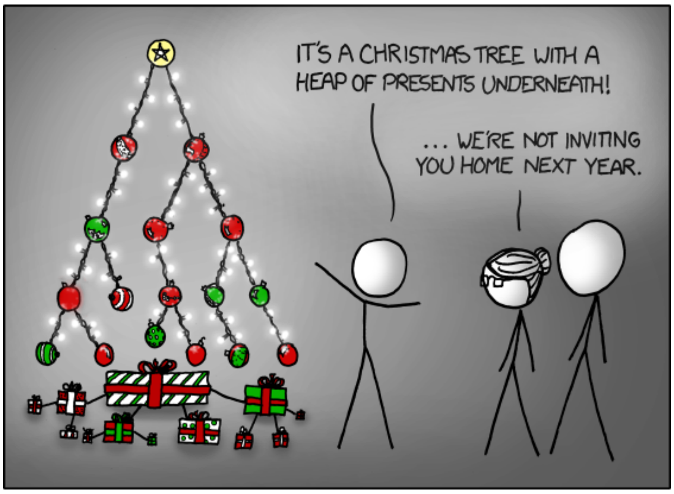
Assignment 3: Inheritance, polymorphism, dynamic binding



Tree Randall Munroe (`http://xkcd.com/835/`)

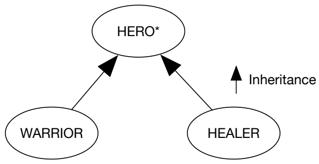
Published: October 25th, 2016
Due: November 26th, 2016

Goals

- Understand polymorphism and dynamic binding.
- Practice inheritance.
- Design and implementation of a 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 the video game “Blades of Glory”.




```
deferred class HERO

feature -- Initialisation
    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 -- Initialisation
    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
```

To do

Given the following variable declarations:

```
hero: HERO
warrior: WARRIOR
healer: HEALER
l: LINKED_LIST [HERO]
```

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: The following code

```
create warrior
warrior.level_up
```

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

Task 1:

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

Task 2:

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

Task 3:

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

Task 4:

```
create warrior.make ("Thor")
create healer.make ("Althea")
create l.make
l.extend (warrior)
l.extend (healer)
across l as h loop h.item.level_up end
```

Task 5:

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

Task 6:

```
create {WARRIOR} hero.make ("Thor")
create {HEALER} hero.make ("Althea")
create l.make
l.extend (hero)
across l as h loop h.item.level_up end
```

Task 7:

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

Task 8:

```
create warrior.make (“Thor")
create healer.make (“Althea")
create l.make
l.extend (warrior)
l.extend (healer)
across l as h loop h.item.do_action (warrior) end
```

Task 9:

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

To hand in

Hand in your answers for the code fragments above.

2 Bags

A *bag* (also called *multiset*) is a generalisation of a set, where elements are allowed to appear more than once. For example, the bag $\{a, a, b\}$ consists of two copies of a and one copy of b. However, a bag is still unordered, so the bags $\{a, b, a\}$ and $\{a, a, b\}$ are equivalent.
In this task you have to implement some features for a linked representation of finite bags. This representation is very similar to a regular singly-linked list, except for the following:

- In addition to the value and the reference to the next cell, each bag cell stores the number of copies of its value (see Figure 1), which is always positive.
- For a given value, at most one cell storing that value should appear in the data structure.
- The *bag* may contain any type of elements. Although, the values stored in the bag need to have the notion of comparison.

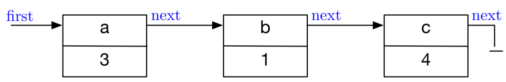


Figure 1: A possible linked representation of a bag of characters $\{a, a, c, c, b, a, c, c\}$.

To do

1. Download `https://drive.google.com/open?id=0B1GMHm59JFjqajFkZXg2VXlBRHc`, unzip it and open linked `bag.ecf`. Open class *LINKED_BAG*.

2. Fill in the implementations of the following features:

   - *add (v: G; n: INTEGER)*, which adds n copies of v to the bag.
   - *remove (v: G; n: INTEGER)*, which removes as many copies of v as possible, up to n. For example, removing one copy of a from the bag $\{a, a, b\}$ will result in a bag $\{a, b\}$, while removing two copies of c from the same bag will not change it.
   - *subtract (other: LINKED_BAG [G])*, which removes all elements of other from the current bag. For example, taking the bag $\{a, a, b\}$ and subtracting $\{a, b, c\}$ from it will yield the bag $\{a\}$.

   Your implementation should satisfy the provided contracts.

3. Add the following features:

   - *min: G*, which gives the minimum element (regardless the number of occurrences of it) from the bag (this is possible since the bag will contain elements that can be compared). For example, the minimum element from the bag $\{3, 1, 1\}$ will result in 1, and the minimum element from the bag $\{a, a, b\}$ will result in a.
   - *max: G*, which gives the maximum element (regardless the number of occurrences of it) from the bag (this is possible since the bag will contain elements that can be compared). For example, the maximum element from the bag $\{3, 1, 1\}$ will result in 3, and the maximum element from the bag $\{a, a, b\}$ will result in b.

   Add the corresponding contracts.

4. To test your implementation, create a class *CARD* (as the one you created for the Card Game in one of the Lab Sessions – notice that you have to implement the notion of order in the cards). Then, in class *TEST* (you can find it in the zip file you downloaded), create a *LINKED_BAG* of cards and implement an algorithm that takes a *LINKED_BAG [CARD]* and returns an ordered *LINKED_LIST [CARD]*.

To hand in

Hand in your code (i.e. files `linked_bag.e`, `test.e` and `card.e`)

3 Board game

The idea is to program a prototype of a board-game. It comes with a *board*, divided into 40 *squares*, and a pair of *dice*; the game can accommodate 2 to 6 *players*. It works as follows:

- All players start from the first square.

- One at the time, players take a turn: roll the dice and advance their respective tokens on the board.
- If a player rolls doubles his token needs to go backwards the number given by one of the dice. For example, if the dice rolled 3 and 3, the player needs to go backwards 3 squares (or remain in the starting point if there are more squares to go backwards than the initial square).
- A player must roll the exact number to reach the final square to win. If the roll of the die is too large the player's token remains in place.
- A round consists of all players taking their turns once.
- Players have money. Each player starts with 50 Rub.
- The amount of money changes when a player lands on a special square:
  - Squares 6, 16, 26, 36 are bad investment squares: a player has to pay 50 Rubs. If the player cannot afford it, he gives away all his money.
  - Squares 9, 19, 29, 39 are lottery win squares: a player gets 100 Rubs.
- The winner is the player with the most money after the first player advances exactly until the 40th square. A draw game (multiple winners) is possible.

To do

Implement the prototype of the board game using the following classes:

- *GAME*: encapsulates the logic of the game (start state, the structure of a round, ending conditions).
- *DIE*: implements one die.
- *PLAYER*: stores the state of each player in the game, his money and performs a turn.

Use class *APPLICATION* as root class of your system, which is responsible for interaction with the user.
Hint: To generate a sequence of random numbers, you can use the class V_RANDOM. Here is an example of how it works:

```
print_random
        -- Print random numbers in range [1 .. 100] until we hit 13
    local
        random: V_RANDOM
    do
        from
            create random
        until
            random.bounded_item (1, 100) = 13
```

```
    loop
        print (random.bounded_item (1, 100))
        io.new_line
        random.forth
    end
end
```

To hand in

Hand in your source code (i.e. files `game.e`, `die.e`, `player.e` and `application.e`)