Student Name:

Student Number:

Midterm Examination
COMP 304B 2004: Object-oriented Design

Examiner: Prof. Hans Vangheluwe — Friday February 20^{th}, 2004

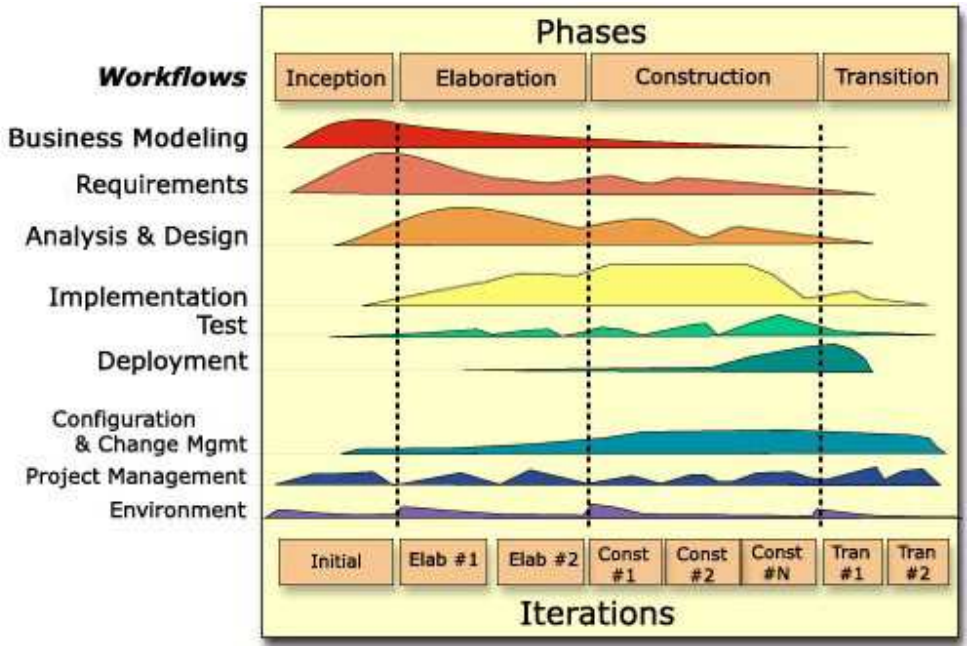
Invigilators: Sadaf Mustafiz, Marc Provost — 14:30 – 15:30

INSTRUCTIONS:

1. Answer all questions directly on the examination paper.
2. No aids of whatever type are permitted.
3. The exam has 7 questions on 7 pages (not including the cover page).
4. Attempt all questions: partial marks are given for incomplete but correct answers. If you make assumptions, mention them explicitly.
5. Numbers between brackets [] denote the weight of each question. The exam is out of a total of 40 points.
6. The midterm counts for 15 % of the total COMP 304 grade.
7. Use the back of the last page as scrap (it will be ignored during grading). The rear of the other pages may be used as extra space to answer questions.

Good luck !

(1) [3]

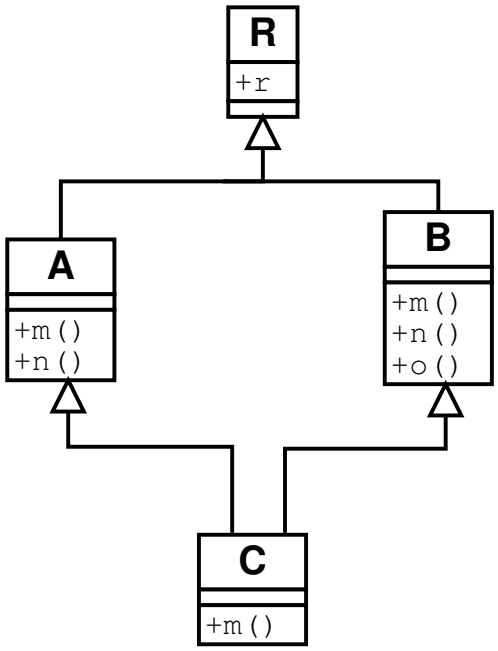


What can we infer from the above figure about the software process used ?

(2) [2]

What are the different types of unit tests ? For each type, give a one-line description.

(3) [6]



1. **[2]** Referring to the above figure, explain the problems with *repeated* and *multiple* inheritance.

2. **[2]** `a, b, c` are instances of `A, B, C` respectively. If the multiple inheritance semantics of Python is used for `class C(A,B)`, which class' method (circle the appropriate one) will be invoked with

(a) `a.m()`	A	B	C	none
(b) `a.n()`	A	B	C	none
(c) `a.o()`	A	B	C	none
(d) `b.m()`	A	B	C	none
(e) `b.n()`	A	B	C	none
(f) `b.o()`	A	B	C	none
(g) `c.m()`	A	B	C	none
(h) `c.n()`	A	B	C	none
(i) `c.o()`	A	B	C	none

3. **[1]** Is the attribute `r` in `R` a part of (the state of) instances of `C` ? YES NO.

4. **[1]** If the attribute `r` in `R` were *private*, would `r` be part of (the state of) instances of `C` ? YES NO.

(4) [2]

What is a *class attribute* and how is it represented in UML ?

(5) [15]

A class `PositiveInteger` has two public attributes: `value` of type `int` and `isZero` of type `enum{True, False}`. The class invariant of `PositiveInteger` states that `value >= 0`.

A class `A` has two public attributes: `a1` and `a2`, both instances of class `PositiveInteger`. The class invariant of `A` states that `a1.value + a2.value > 10`. `A` also has a public operation `update` which takes two arguments `arg1` and `arg2`, both of type `int`. `update` has not return value but it does change `A`'s attributes `a1` and `a2`. `update` has as pre-condition which states that `arg1` and `arg2` must both be larger than 2. `update` has as post-condition which states that `a1.value > 10` and `a2.value > 100`.

A class `B` inherits from `A`. In `B`, a private attribute `b` of type `int` is declared. `B` also defines a public operation `update` with *exactly* the same signature as the `update` declared in class `A`.

- **[2]** Draw the class diagram for the above (including all information such as pre/post-conditions).

- **[2]** What is the state-space of (instances of) each of the classes in the class diagram ?

- **[1]** What is the name given to the relationship between the operation `update` in classes `A` and `B` ?

- In good OO design, what should be the relationship between (include concrete examples which satisfy the required relationships):

  1. **[2]** the state space of class `B` and that of class `A`;

  2. **[4]** the pre/post-conditions of `update` of class `B` and of class `A`. Under which name are the relationships also known (in type theory);

3. **[2]** the invariants of class `B` and of class `A`.

- **[2]** What is the principle of closed behaviour in good OO design ? Illustrate by means of a concrete example involving the classes `A` and `B`.

(6) [8]

Draw a *class diagram* for the following (including reasonable types for attributes):

A `Company` has at least one `Department`. A `Department` is a "logical" part of a `Company`. A `Company` also has at least one `Office`. An `Office` is a "physical" part of a `Company`. Both an `Office` and a `Department` belong to exactly one `Company`. Over time, the number of `Department`s and `Office`s in a `Company` may change. A `Department` has its physical `Location` in one or more `Office`s. Multiple `Department`s may be located in the same `Office`. The `Company` may even have `Office`s without `Department`s located there. A `Department` can have any number of `daughter Department`s.

A `Department` has a `name`. An `Office` has a `address` and a `phonenr`. An `Office` consists of one or more `Building`s. A `Building` has one or two `frontDoor`s and 0 or one `rearDoor`, both of which are `Door`s. A `Door` has a `width` and a `height`. A `building` must be able to reference its doors, but not the other way around.

`Headquarters` is a special kind of `Office`. A `Headquarters` may or may not have a company flag over its entrance. This is denoted by `hasCompanyFlag`.

A `Person` has an `age` and a `name`. `Man` and `Woman` are the only possible types of `Person`s. A `Person` has a `birthData` and a `name`. Men and women can be related through `Marriage`. *Exactly* one Man and one Woman who are married can together have any number of `children`, who are `Person`s.

Every `Department` has an *ordered* collection of one or more `members` who are `Employee`s. A `Department` must be able to reference all its `members`, but not the other way around. An `Employee` is a `Person`. An `Employee` has an `id` and a `title`. An `Employee` can be a member of any number of `Department`s. A `Manager`

is a special kind of `Employee`.

(7) [4]

Draw both a *class diagram* **[1]** and a *collaboration diagram* **[3]** (compatible with one another) for two `Client` objects `client1` (first) and `client2` (later) registering themselves by means of an appropriate `register` message with a `Server` object. Some time later, the server will receive a `warning("HIGH")` message from an instance of the `TemperatureMonitor` class. The parameter `"HIGH"` denotes the severity level of the warning. Reception of the `warning` prompts the server to *callback* both clients in the order they registered themselves and invoke their `do_it` operation. While doing so, the server will make it possible for the client to know *which object* called its `do_it` operation. The `do_it` operation in *both* objects `client1` and `client2` invokes the `help` operation (with 0 arguments) in a unique object `helperSingleton`, an instance of class `Helper` to do

whatever it is supposed to do.