Programación orientada a objetos. Examen febrero 2002.
Martes 5 de febrero de 2002.

1. a. Definan en forma breve y precisa que es el **estado** de un objeto. ¿Qué pueden decir del estado de los objetos de las clases que tienen sólo métodos de clase? Justifiquen la respuesta.

   b. Definan en forma breve y precisa que es el **comportamiento** de un objeto. ¿Tiene sentido que haya objetos sin comportamiento? Justifiquen la respuesta.

   c. ¿Todos los objetos de la misma clase tienen el mismo estado? Justifiquen la respuesta.

   d. ¿Todos los objetos de la misma clase tienen el mismo comportamiento? Justifiquen la respuesta.

   10 puntos

2. Se suele afirmar que en Smalltalk todo es un objeto. La clase de un objeto es también un objeto de la clase `Behavior`. Los mensajes que se envía a un objeto se llaman métodos de instancia y los que se envía a la clase de un objeto se llaman métodos de clase. Pero los métodos de clase de cualquier clase son métodos de instancia de la clase `Behavior`. Supongan que existen los siguientes métodos de Smalltalk:

Selector	Clase	Descripción
`class`	`Object`	Retorna la clase del receptor.
`subclass`	`Behavior`	Retorna la superclase del receptor.
`sourceCodeAt: aSymbol`	`Behavior`	Retorna una `String` con el código fuente del método llamado `aSymbol` en el receptor o `nil` si el receptor no implementa el método.

   Programen en Smalltalk un método de instancia al que puedan responder todos los objetos, que retorne el código fuente dado el nombre de un método. Elijan ustedes la clase en la que implementarlo y la signatura del método.

   10 puntos

3. Imaginen un lenguaje de programación orientado a objetos que es similar a Smalltalk -llamado Xmalltak- en el que se declara el tipo de las variables así:

```
| a: String b: Collection |
a := String new.
b := Collection with: a.
```

   Las asignaciones incorrectas son detectadas al evaluarlas con **Do it** o **Show it**. Xmalltalk considera que una asignación es incorrecta siguiendo el criterio de sustitución que se vio en clase.

   Al evaluar el siguiente código

```
| a: String |
a := Integer new.
```

   Xmalltalk inserta el texto "`assignment missmatch`" después del símbolo de asignación "`:=`" como sucede en Smalltalk con el texto "`invalid receiver`" o "`unfinished comment`":

```
|a: String |
a := assginment missmatch Integer new.
```

   Los mensajes con selector no implementado en ningún método del receptor son detectados al evaluarlos con Do it o Show it. Xmalltalk usa encadenamiento dinámico para la resolución de métodos en tiempo de ejecución.

   Al evaluar el siguiente código

```
|a: String |
...
a factorial
```

Xmalltalk inserta el texto "`invalid receiver`" antes del selector:

```
|a: String |
...
a invalid receiver factorial
```

Sean ahora dos clases en Xmalltalk definidas así:

```
Object subclass: #Alfa
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !
! Alfa methods !
isAlfa
  ^true !

Alfa subclass: #Beta
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: '' !
! Beta methods !
isAlfa
  ^false !

isBeta
  ^true !
```

Escriban un fragmento de código Xmalltalk que evaluado con Show it :

a. Retorne `true`.

b. Retorne `false`.

c. Ocurra un error "`assignment mismatch`".

d. Ocurra un error "`invalid receiver`".

El fragmento de código debe tener la forma:

```
| a: «Alfa|Beta» b: «Alfa|Beta»|
a := «Alfa|Beta» new.
b := «Alfa|Beta» new.
^a «isAlfa|isBeta» and: [b «isAlfa|isBeta»].
```

20 puntos

4. Implementen un método **`fibonnacci`** en la clase apropiada –que deberán decir cuál es– para calcular el valor de la secuencia Fibonnacci que se define recursivamente así: para un número natural *n*:

   $fibonnacci(0) \equiv fibonnacci(1) \equiv 1$
   $fibonnacci(n + 1) \equiv fibonnacci(n) + fibonnacci(n - 1) \quad \forall n > 1$

   ¿Qué precondición agregarían a ese método? Escríbanla usando el método `assert` de la clase `Context` tal como se hizo en clase.

10 puntos

5. En la descomposición orientada a objetos vemos el mundo como una colección significativa de agentes autónomos que colaboran para mostrar un nivel más elaborado de comportamiento. Para que esto funcione son necesarios los contratos y las responsabilidades. Definan **contratos** y **responsabilidades** y justifiquen la afirmación anterior.

10 puntos

6. Consideren la clase Persona implementada en Smalltalk de la siguiente manera:

```
Object subclass: #Persona
  instanceVariableNames:
    'nombre '
  classVariableNames: ''
  poolDictionaries: '' !
```

```
!Persona methods !

nombre
    "Retorna el nombre de esta persona."
    ^nombre !

nombre: aString

    "Asigna el nombre de esta persona con el argumento aString."
    Nombre := aString !
```

Consideren la clase `Asalariado` implementada en Smalltalk de la siguiente manera:

```
Persona subclass: #Asalariado
  instanceVariableNames:
    'salario '
  classVariableNames: ''
  poolDictionaries: '' !

!Asalariado methods !

salario
    "Retorna el salario de esta persona."
    ^salario !

salario: anInteger

    "Asigna el salario de esta persona con el argumento anInteger."
    salario := anInteger !
```

Usando por lo menos la clase `Persona` provista y mediante **composición** y **delegación** –en lugar de herencia- programen nuevamente la clase `Asalariado`. Indiquen claramente dónde está la composición y dónde la delegación.

20 puntos

7. Supongan que sobre la clase `Persona` del ejercicio anterior existe una invariante que establece que el nombre nunca puede quedar en blanco –ni contener `nil` ni una instancia de `String` vacía-. Introduzcan todas las modificaciones que consideren necesarias para implementar esa invariante. Usen el método `assert` de la clase `Context` tal como se hizo en clase.

10 puntos

8. En tiempo de ejecución un objeto recibe un mensaje cuyo selector corresponde a un método que la clase de ese objeto no tiene. ¿Qué ocurre? ¿Porqué? ¿Es posible que esta situación ocurra usando encadenamiento estático? ¿Porqué?

10 puntos