Trabajo Práctico 3
Programación Orientada a Objetos

Paradigmas de Lenguajes de Programación — 1er cuat. 2016

Fecha de entrega: 21 de junio

Introducción

Este trabajo consiste en modelar lógica proposicional en Smalltalk, de modo de poder construir fórmulas y evaluarlas bajo distintas valuaciones.

Una fórmula proposicinal puede ser:

- una variable proposicional cuyo nombre es un símbolo (ej. `#P`)
- la negación de una fórmula
- una conjunción entre dos fórmulas
- una disyunción entre dos fórmulas
- una implicación entre dos fórmulas

Se deberán implementar las clases necesarias para poder modelar todos estos tipos de fórmulas (eventualmente puede ser necesario crear una o más clases adicionales para abstraer características comunes).

Las valuaciones se representarán como conjuntos de símbolos, donde cada símbolo corresponde al nombre de una variable proposicional que se considera verdadera en esa valuación.

Por ejemplo, el conjunto `Set with: #P with: #Q with: #R` representa la valuación donde las variables proposicionales llamadas `#P`, `#Q` y `#R` son verdaderas (y todas las demás son falsas).

El trabajo estará guiado a través de una serie de tests inspirados en la técnica *Test Driven Development (TDD)*[1].

Por consiguiente, se recomienda implementar el mínimo código razonable para lograr ir pasando los tests de manera progresiva, manteniendo en funcionamiento los tests previos. Cada ejercicio irá acompañado de un test que tenga la información necesaria para resolverlo.

[1] Tomar la metodología presentada como una introducción a la técnica. Para más detalles, recomendamos la materia POO de Hernán Wilkinson.

Ejercicios a desarrollar

Ejercicio 1

Implementar las clases necesarias (con sus respectivas inicializaciones) para poder representar todas las fórmulas proposicionales. En este punto la clase `PropositionalFormula` solo cumplirá el objetivo de poder reconocer a todas las fórmulas como tales. Al finalizar este ejercicio, deberán poder pasar los tests de la clase `Ej1KindOfTest`.

Ejercicio 2

Implementar un mecanismo para obtener los nombres de todas las variables proposicionales de una fórmula (como conjunto). Al finalizar este ejercicio, deberán poder pasar los tests de la clase `Ej2AllPropVarsTest`.

Ejercicio 3

Implementar en las clases que correspondan el método `value:`, que recibe como parámetro una valuación (representada como se describió anteriormente) y devuelve `true` o `false` según si dicha valuación satisface o no la fórmula.

Dado que todas las operaciones binarias responderán a este mensaje de manera similar (excepto por el operador utilizado), se deberá abstraer este comportamiento común para evitar repetir código. Para esto se recomienda enviar el mensaje `sendTo:` a una instancia de la clase `Message`. Notar que `true` y `false` saben responder los mensajes binarios `&`, `|` y `==>`.

Al finalizar este ejercicio deberán pasar el test de la clase `Ej3ValueTest`, y podrán decir que usaron metaprogramación.

Ejercicio 4

Ya habrán notado lo molesto que es escribir fórmulas usando los constructores de sus clases (si no lo notaron, escriban algunos tests adicionales para los ejercicios anteriores).

A partir de este ejercicio, se deberá poder construir fórmulas de manera más sencilla, utilizando los mensajes `&`, `|`, `==>` y `not`. Al finalizar este ejercicio deberán pasar el test de la clase `Ej4PrettyFormulaConstructionTest`.

Ejercicio 5

Ya podemos construir fórmulas de manera más cómoda, pero todavía no podemos verlas bien con el comando "Print it". En este ejercicio, deberán implementar los métodos `asString` y `printString` (que deberían devolver lo mismo). Las condiciones para convertir fórmulas a `String` son las siguientes:

- Para las variables, devolver su nombre en forma de `String`.
- La negación de una fórmula se convertirá en un string de la foma '¬A', donde A es la representación textual de la fórmula negada (entre paréntesis si se trata de una operación binaria).

- Las conjunciones, disyunciones e implicaciones se traducirán a strings de la forma 'A & B', 'A | B' y 'A ==> B' respectivamente, con las mismas consideraciones para A y B que en el caso de las negaciones.

Es importante no repetir código ni abusar de los condicionales (en este caso, los condicionales no deberían ser necesarios). Al finalizar este ejercicio, deberán poder pasar los tests de la clase `Ej5StringConversionTest`.

Ejercicio 6

Tanto para hacer buenos tests como para poder definir colecciones de fórmulas, es necesario poder comparar fórmulas por igualdad. Deberán definir los métodos `=` y `hash` de manera tal que puedan aplicarse correctamente a todas las fórmulas.

Utilizaremos la igualdad sintáctica y no semántica (determinar si dos fórmulas proposicionales son semánticamente equivalentes es un problema NP-completo). Al finalizar este ejercicio deberán pasar los tests de la clase `Ej6EqualityTest`.

Pista: dado que estos mensajes suelen ser necesarios, la mayor parte de los objetos ya saben responderlos.

Ejercicio 7

Finalmente, vamos a repasar un poco lo que vimos en resolución y pasar las fórmulas a forma normal negada.

Como primer paso para esto, deberán implementar el método `negate`, para negar una fórmula sin introducir negaciones por fuera de otros operadores. Para esto, pueden utilizar las siguientes equivalencias lógicas (expresadas con notación del TP):

$$\begin{aligned} \neg\neg A &\equiv A \\ \neg(A \ \&\ B) &\equiv \neg A \mid \neg B \\ \neg(A \mid B) &\equiv \neg A \ \&\ \neg B \\ \neg(A ==> B) &\equiv A \ \&\ \neg B \end{aligned}$$

Luego deberán implementar `toNNF` (del inglés *Negated Normal Form*), que traduce una fórmula a su forma normal negada.

Una vez concluido este ejercicio deberán poder pasar todos los tests.

Pautas de entrega

El entregable debe contener:

- un archivo `.st` con todas las clases implementadas
- versión impresa del código, comentado adecuadamente (puede ser el propio .st sin los tests)
- **NO** hace falta entregar un informe sobre el trabajo

Se espera que el diseño presentado tenga en cuenta los siguientes factores:

- definición adecuada de clases y subclases, con responsabilidades bien distribuidas
- uso de polimorfismo para evitar exceso de condicionales
- intento de evitar código repetido utilizando las abstracciones que correspondan

Consulten todo lo que sea necesario.

Consejos y sugerencias generales

- Lean al menos el primer capítulo de *Pharo by example*, en donde se hace una presentación del entorno de desarrollo.
- Explorar la imagen de Pharo suele ser la mejor forma de encontrar lo que uno quiere hacer. En particular tengan en cuenta el buscador (`shift+enter`) para ubicar tanto métodos como clases.
- No se pueden modificar los test entregados, si los hubiere, aunque los instamos a definir todos los tests propios que crean convenientes.

Importación y exportación de paquetes

En Pharo se puede importar un paquete arrastrando el archivo del paquete hacia el intérprete y seleccionando la opción "`FileIn entire file`". Otra forma de hacerlo es desde el "`File Browser`" (botón derecho en el intérprete > `Tools` > `File Browser`, buscar el directorio, botón derecho en el nombre del archivo y elegir "`FileIn entire file`").

Para exportar un paquete, abrir el "`System Browser`", seleccionar el paquete deseado en el primer panel, hacer click con el botón derecho y elegir la opción "`FileOut`". El paquete exportado se guardará en el directorio `Contents/Resources` de la instalación de Pharo (o en donde esté la imágen actualmente en uso).