Implementación de Lenguajes Orientados a Objetos

Traducción de código Java a SimpleSem

Lenguajes de Programación
Julio de 2025

Ludmila Prolygin

Tabla de contenidos

Para la resolución completa del proyecto, previo a la traducción, se hizo un análisis pormenorizado de la lógica del código provisto en Java a fines de simplificar el desarrollo.

Sobre el uso de herramientas y fuentes externas. Fue utilizado ChatGPT solo con el propósito de obtención de casos de prueba para la verificación de la traducción, a fines de ampliar los casos de prueba iniciales y posibilitar el análisis de casos potencialmente no evaluados.

Análisis y explicación

Se incluye una etiqueta `llamador` con el propósito de actualización de registros Actual y Libre de la memoria de Datos, para no repetir las instrucciones al final de la traducción de los métodos.

Se optó por no incluir una etiqueta `mod` puesto que la resolución podía realizarse en una instrucción con múltiples accesos a memoria. En caso de crear la etiqueta y realizar el salto para resolverlo, implica sobrecarga de memoria de datos, creando un registro de activación extra, además de los accesos de memoria utilizados.

Esquemas

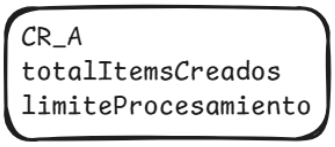
Virtual Tables (VTs)

```
VT_A
[A] ref procesarLote()
[A] ref obtenerInformacion(int tipoInfo)
[A] ref obtenerValorUnitario()
[A] ref verificarStock()
```

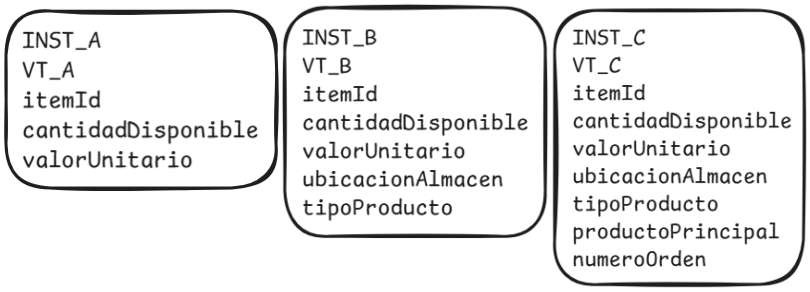
```
VT_B
ref VT_A
[B] ref procesarLote()
[B] ref obtenerInformacion(int tipoInfo)
[B] ref obtenerValorUnitario()
[A] ref verificarStock()
[B] ref manejarPedido(int cantidadSolicitada)
[B] ref manejarPedido(int cantidadSolicitada, int prioridad)
```

```
VT_C
ref VT_B
[B] ref procesarLote()
[B] ref obtenerInformacion(int tipoInfo)
[B] ref obtenerValorUnitario()
[A] ref verificarStock()
[B] ref manejarPedido(int cantidadSolicitada)
[B] ref manejarPedido(int cantidadSolicitada, int prioridad)
[C] ref inicializarProductoPrincipal()
[C] ref cumplirOrdenCompleja(int cantidadNecesaria)
```

Class Records (CRs)



Class Instance Records (INSTs)



Registros de Activación (RAs)

Clase A

RA_constructorA
punteroRetorno:
enlaceDinamico:

RA_procesarLoteA
punteroRetorno:
enlaceDinamico:
this:
itemsProcesados:
i:
stock:

RA_obtenerInformacionA
punteroRetorno:
enlaceDinamico:
this:
tipoInfo:

RA_obtenerValorUnitarioA
punteroRetorno:
enlaceDinamico:
this:

RA_verificarStockA
punteroRetorno:
enlaceDinamico:
this:

Clase B

RA_constructorB
punteroRetorno:
enlaceDinamico:

RA_procesarLoteB
punteroRetorno:
enlaceDinamico:
this:
resultadoBase:
ajuste:

RA_obtenerInformacionB
punteroRetorno:
enlaceDinamico:
this:
tipoInfo:

RA_obtenerValorUnitarioB
punteroRetorno:
enlaceDinamico:
this:

RA_manejarPedido1B
punteroRetorno:
enlaceDinamico:
this:
cantidadSolicitada:

RA_manejarPedido2B
punteroRetorno:
enlaceDinamico:
this:
cantidadSolicitada:
prioridad:
costo:

Clase C

RA_constructorC
punteroRetorno:
enlaceDinamico:
ordenId:

RA_inicializarProductoPrincipalC
punteroRetorno:
enlaceDinamico:
this:

RA_cumplirOrdenComplejaC
punteroRetorno:
enlaceDinamico:
this:
cantidadNecesaria:
itemsFaltantes:
vecesProcesado:
costoTotal:
procesadoAhora:

Clase Principal

RA_main
punteroRetorno:
enlaceDinamico:
solicitudInicial:
prioridadOrden:
gestorPrincipal:

Mecanismo de iteración de cumplirOrdenCompleja de la clase C

Para la traducción de `cumplirOrdenCompleja` de la clase C, se tuvo en cuenta la lógica de repetición de la sentencia propuesta. Dado que se utiliza una estructura de tipo `do <...> while (condicion)`, el código se ejecuta siempre al menos una vez. Por este motivo, el salto condicional se incluye al final de las sentencias contenidas en la estructura.

Posicionar el la condición de corte de la iteración al final del código es la principal diferencia puesto que, en `procesarLote` de la clase A, la condición de corte está ubicada antes de las sentencias a repetir. En este caso, la decisión fue tomada considerando que la estructura de `procesarLote` es de tipo `while (condicion) do <...>` por lo que, podría ser que las instrucciones allí contenidas, no se ejecuten ninguna vez.

Otra diferencia relevante fue la traducción de las condiciones; `cumplirOrdenCompleja` utiliza una logica con cortocircuito (&&) mientras que, `procesarLote` no (&). En el apartado Expresiones booleanas se ahondará más sobre las diferencias entre estas expresiones booleanas.

Mecanismo de selección múltiple

Para la resolución del mecanismo de selección múltiple propuesto en `procesarLote` de la clase B (`switch`), se incorporó un salto condicional por cada valor posible de selección (`case`). Es una principal diferencia respecto de la utilización de la sentencia if con bloques if anidados puesto que, en este último caso, se permite hacer ejecuciones con menor cantidad de saltos, pudiendo hacer uso del PC (Program Counter). Además, con la sentencia condicional utilizada, fue necesario contemplar el momento de finalización del caso para realizar un salto adicional de salida del mecanismo propuesto.

Esto permite entender la importancia de incluir la sentencia `break;` al finalizar cada caso.
La resolución fue posible debido a que los tipos a comparar eran enteros y SimpleSem soporta nativamente esta situación. En caso de que el tipo fuera más complejo, sería necesario acceder a memoria H para comparar los campos que describen a la instancia a comparar.

Expresiones booleanas

Para la traducción de las expresiones booleanas sin cortocircuito, basta con utilizar el operador (`&` o `|`) provisto por SimpleSem. Por ejemplo: condición de corte de la sentencia de repetición condicional `while` del método `procesarLote` de la clase A.
Para la traducción de las expresiones booleanas con cortocircuito, no es correcto utilizar los operadores provistos por SimpleSem. En estos casos, el análisis del valor de verdad de la segunda parte de la expresión, está directamente relacionado con el valor de verdad adoptado por la primera parte. Para resolverlo, es necesario tener presente el valor de las tablas de verdad y analizar cuáles son los casos relevantes.

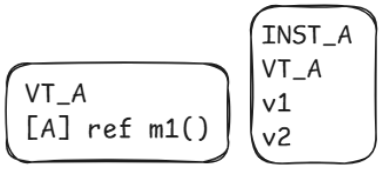
A	B	and	or
t	t	t	t
t	f	f	t
f	t	f	t
f	f	f	f

Es visible que, el resultado de `and` siempre es falso a menos que ambos miembros del análisis sean verdaderos y, el resultado de or es siempre verdadero a menos que ambos miembros del análisis sean falsos. Con esta lógica subyacente, es posible determinar que, en el caso de utilizar and con cortocircuito (&&), el análisis del segundo miembro solo debe ser realizado cuando el primero sea verdadero; cualquiera sea el valor del segundo miembro, si el primero es falso, toda la expresión será falsa. De forma análoga, en el caso de utilizar or con cortocircuito (`||`), el análisis del segundo miembro solo debe ser realizado cuando el primero sea falso; cualquiera sea el valor del primer miembro, si el segundo es verdadero, toda la expresión será verdadera.
Por este motivo, a diferencia de la traducción de la expresión condicional sin cortocircuito, la traducción tiene al menos un salto condicional más, según el operador binario booleano empleado: según el valor de verdad de la primer parte del análisis, se hace un salto a la evaluación de la segunda parte del análisis o no.

- &&. Si el primer miembro es verdadero, se realiza un salto a la evaluación del segundo miembro y allí se determina si el salto debiera ser al código del `then` o del `else`. Caso contrario, el salto se realiza al `else`.
- `||`. Si el primer miembro es falso, se realiza un salto a la evaluación del segundo miembro y allí se determina si el salto debiera ser al código del `then` o del `else`. Caso contrario, el salto se realiza al `then`.

Opcionales

Opcional 1 - chequeo de nulos

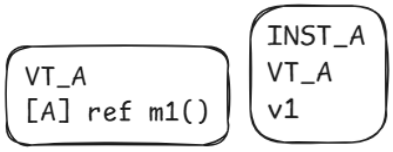


RA_constructorA	**RA_m1**	**RA_main**
punteroRetorno:	punteroRetorno:	punteroRetorno:
enlaceDinamico:	enlaceDinamico:	enlaceDinamico:
	this:	oA:

Cada vez que se crea un objeto, se modifica el valor de PO (registro que apunta al primer lugar de heap disponible para asignar una nueva instancia).
Por este motivo, para controlar que el objeto esté correctamente instanciado, se considera que:

- PO no puede ser 0 (cero). En caso de serlo, no hay ningún objeto creado, por lo que la variable a la que se quiere enviar el mensaje, no está instanciada.
- PO debe ser mayor que el valor que está almacenado en el lugar de la variable en el registro. Si la variable apunta a una posición de heap más grande que PO, ese valor no es consistente con los objetos creados de forma efectiva. Posiblemente el valor allí almacenado sea información persistente de registros de activación que ya fueron destruidos.
- Por la implementación actual, podría suceder que quiera enviarse un mensaje a una variable no ligada, pero de igual forma se satisfaga el valor del PO asociado. Esto supone un error, pero está relacionado al manejo de memoria.

Opcional 2 - asignación condicional



RA_constructorA	**RA_m1**	**RA_main**
punteroRetorno:	punteroRetorno:	punteroRetorno:
enlaceDinamico:	enlaceDinamico:	enlaceDinamico:
x:	this:	o1:
y:		o2:

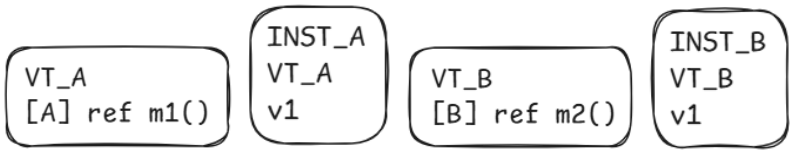
Las asignaciones condicionales en Java son de forma
`variable = (condicion) ? valor_si_verdadero : valor_si_falso`

Esto es equivalente a

```
if(condicion)
      variable = valor_si_verdadero
else
      variable = valor_si_falso
```

La traducción fue realizada a partir del código Java equivalente, con los saltos condicionales adecuados.

Opcional 3 - llamadas encadenados



RA_constructorA
punteroRetorno:
enlaceDinamico:
x:

RA_constructorB
punteroRetorno:
enlaceDinamico:

RA_m1
punteroRetorno:
enlaceDinamico:
this:

RA_m2
punteroRetorno:
enlaceDinamico:
this:
o1:

RA_main
punteroRetorno:
enlaceDinamico:
obj:

Para la implementación de llamadas encadenadas se tuvieron en cuenta los RA (registros de activación), ligaduras dinámicas de código e instanciación de objetos; el acceso coherente y correcto de memoria es el punto principal para la resolución de este escenario.
Para la traducción, fue dividido el mensaje según los receptores adecuados de los mensajes; en el código `obj.m2().m1()` provisto, obj recibe el mensaje `m2` y retorna un objeto que es quien recibe el mensaje `m1()`.
Por este motivo, primeramente se crea el RA de m2 considerando que tiene un retorno (se incluye un corrimiento del registro Libre). Una vez finalizado, en vez de volver a acomodar el valor del registro Libre, se crea el RA asociado a m1 justo debajo (contemplando que también retorna un resultado). De esta manera, se posibilita una gestión de objetos más simple; como el objeto resultante de `obj.m2()` no se almacena en ningún lugar, sino que se utiliza. El seteo de this en el RA de m1, hace uso del lugar de memoria D referenciado con `Libre-2` (retorno de `obj.m2()`) y, `Libre-1`, contendrá el valor de retorno de la llamada completa (`obj.m2().m1()`).

Repositorio

El repositorio del proyecto está disponible en el siguiente enlace:
https://github.com/ludmilaprolygin/ProyectoLP