Persistencia en objetos

por Leonardo Gassman, Fernando Dodino, Claudio Fernández

Versión 3.0

Julio 2016

Índice

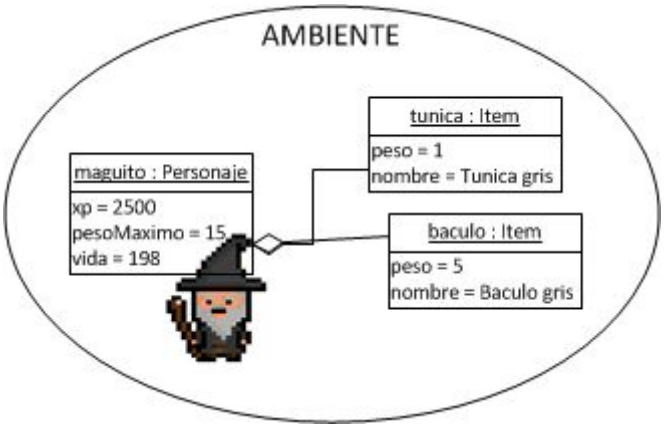
1 Noción de persistencia

1.1 Estado de una aplicación

La concepción más naive de una aplicación es la de un proceso que es ejecutado por el sistema operativo y tiene una determinada área de memoria asignada sobre la cual puede operar y almacenar estado.

Particularmente en los paradigmas imperativos (entre los cuales contamos a la programación en objetos) el estado ocupa un papel fundamental. En la práctica un sistema puede ser observado como una transformación continua de dicho estado en respuesta a los distintos estímulos externos.

Ejemplo:
Tengo un pequeño sistema online. Es un juego, en donde tengo mi personaje quien posee un inventario y una cantidad de vida y experiencia. Este muy sencillo sistema lo tengo modelado en objetos.



*Mi sistema tiene **estado**:*

- *que mi personaje tenga referencia a números (como la cantidad de "vida") es parte del estado de mi sistema.*
- *que mi personaje tenga relaciones con otros objetos que forman parte de su inventario es parte del estado de mi sistema.*
- *que mi personaje exista (sea un objeto que tenga una identidad) también es parte del estado de mi sistema.*

El comportamiento de mi sistema muy probablemente producirá modificaciones en dicho estado. Mi personaje ganará o perderá vida, experiencia (xp) e items. ¡O incluso podría desaparecer completamente!

***Nota**: Volveremos sobre este ejemplo muchas veces en este documento. Por simplicidad de ahora en más dibujaremos las instancias como simples círculos pero en rigor nos estamos refiriendo a lo mismo.*

Nuestro estado vive en nuestros objetos. Nuestros objetos viven dentro de un ambiente. Dicho ambiente existe en la memoria asociada al proceso de que corre la máquina virtual. En muchos sistemas el estado ocupa un papel fundamental, y de ello surgen necesidades comunes:

Es deseable que el estado sobreviva a la volatilidad de la memoria y pueda ser recuperado en ejecuciones posteriores.

Mientras el proceso se encuentre corriendo, el sistema operativo garantizará la disponibilidad del estado. Pero si el proceso terminase la memoria asociada al mismo será liberado y estado entonces destruido.

Dependiendo del ciclo de vida de nuestra aplicación la muerte del proceso puede ser algo natural (un proceso batch que empieza, trabaja sobre un conjunto de datos y termina) o algo fortuito (un corte de luz apaga nuestro servidor de aplicaciones)

A veces se vuelve necesario garantizar que el estado sea preservado y ejecuciones posteriores de nuestra aplicación puedan seguir trabajando con el.

¿Es esto un requerimiento para todos los sistemas? De ningún modo. Esto es simplemente una generalización. Algunos sistemas no tienen estado, su estado no varía de formas significativas (p.e: un calculadora) o su estado puede ser recreado completamente en cada ejecución.

Es deseable poder trabajar con estados más grandes que la capacidad física de la memoria.

Conforme al estado de nuestra aplicación crezca el tamaño de nuestro ambiente crecerá. Existen varias limitaciones tecnológicas asociadas a ambientes muy grandes. La primera limitación es la física: nuestra memoria es limitada y relativamente escasa. Si bien el sistema operativo hace posible que un proceso reserve más memoria que la disponible físicamente (memoria virtual) generalmente las máquinas virtuales (por ejemplo, la jvm de java) no están optimizadas para trabajar con espacios de memoria tan grandes.

Otra característica casi general es que, en un momento determinado, sólo un subconjunto del estado se encontrará en uso. Entonces una solución posible a esta problemática es mantener sólo ese subconjunto en memoria y el resto en algún otro medio persistente del cual pueda ser recuperado oportunamente si es necesario.

Continuando con el ejemplo anterior. Es altamente probable que nuestro sistema tenga muchos personajes de los cuales solo se encontrará activa una pequeña fracción (probablemente muchos otros pertenezcan a jugadores que no estén conectados en este momento).

1.2 ¿Qué significa persistir?

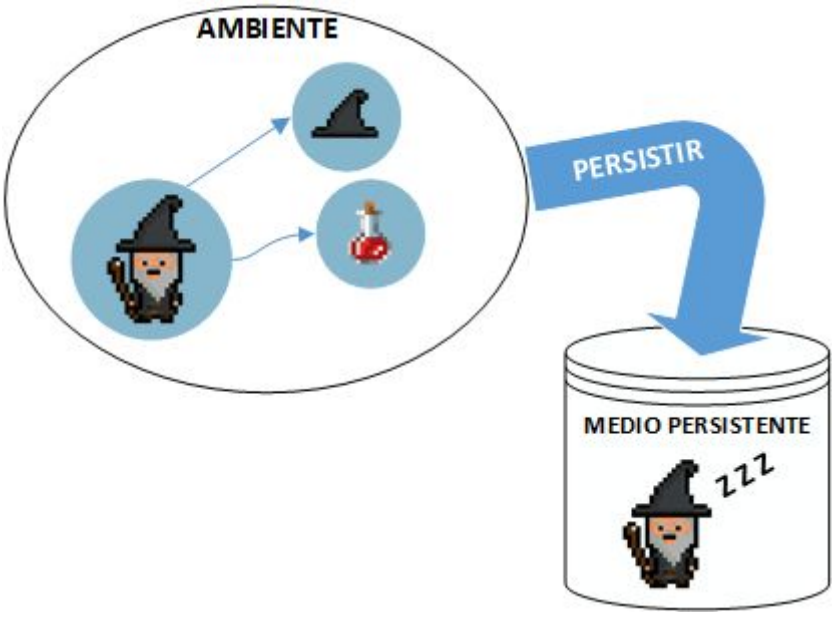
Entendemos entonces como persistencia a la capacidad que una aplicación posee para hacerle frente a una o varias de las necesidades anteriores.

La persistencia implica a priori los siguientes conceptos:

- **Base de datos**: es el conjunto de datos persistidos de la aplicación.
- **Medio Persistente**: Es el soporte físico donde se almacenan los datos, puede ser un disco, un array de discos, una memoria flash o un cluster de máquinas.

¿Cómo es que lucirán mis objetos en ese medio persistente?

Eso todavía no importa. Por ahora, mantengamos el abstracto. Mi maguito puede, de alguna forma, ser guardado en algún medio el cual se garantiza que sobrevivirá a lo que viva mi proceso / ambiente en memoria.



Una complicación que si es necesario notar y es inherente a la persistencia es que, desde este momento, existirán dos copias de nuestro maguito. La primera vive en nuestro ambiente (objetos) y la segunda está “dormida” en un medio persistente. Cómo mantenerlas sincronizadas y qué tan transparente es esa sincronización dependerá de las decisiones de diseño que tomemos más adelante.

1.3 Decisiones de diseño

Algunas preguntas que nos surgirán cuando persistamos son:

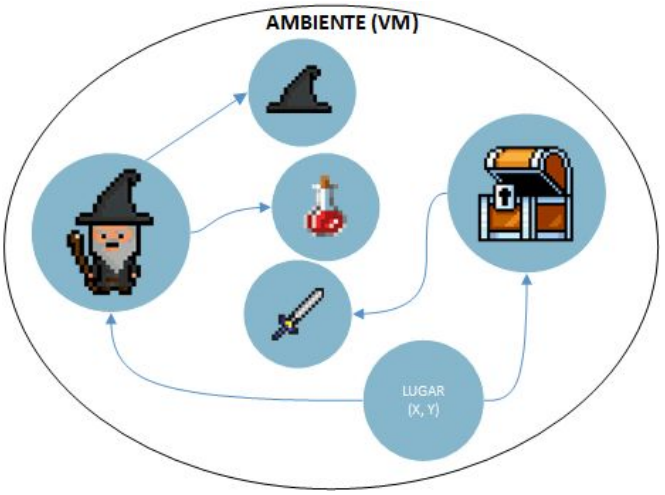
- ¿Qué datos guardar?
- ¿Con qué formato/estructura?
- ¿En qué medio?

- ¿Cuándo guardar?
- ¿Cuándo consultar?
- ¿Cómo consultar?
- ¿Soportaremos actualizar o eliminar los datos? ¿Cómo?
- ¿Cómo mantener la consistencia?

2 Medios persistentes

En este punto nos enfocaremos en el cómo y dónde guardar nuestros objetos. Deberemos tomar la decisión no trivial de elegir qué tipo de estructura de datos tendrá nuestro medio persistente y analizar qué complejidades pueden surgir de adaptar nuestro modelo de objetos a la misma.

Para ello es útil recordar una vez más cómo lucen los objetos dentro de nuestro ambiente. Los mismos se organizan como un grafo dirigido (cada objeto es un nodo y cada referencia una relación dirigida). Este grafo siempre es conexo. A efectos prácticos no tiene objetos ni subgrafos separados, ya que los mismos representan basura y no son significativos (de hecho, serán eliminados eventualmente por el garbage collector)



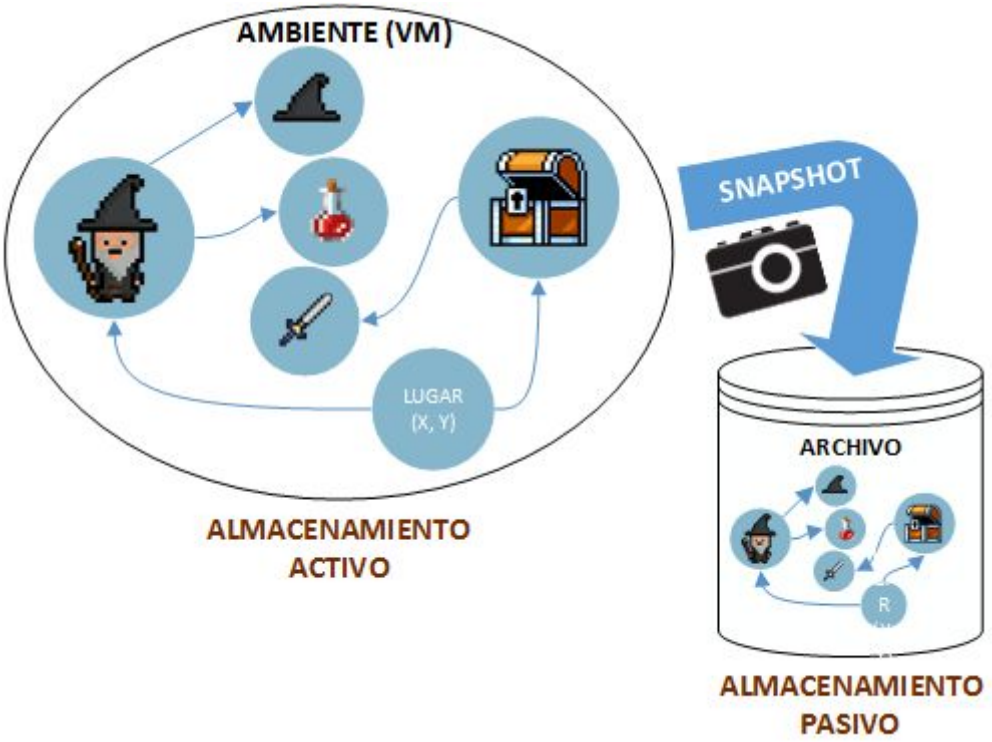
Ampliamos un poco el ejemplo anterior. Nuestro personaje maguito tiene un inventario con su sombrero y una poción. Existe también un cofre con una espada dentro. Tanto el maguito como el cofre se encuentran en una localización puntual del mapa.

Las alternativas más comunes a la hora de persistir nuestros objetos serán:

- Snapshot (foto) del ambiente completo
- Serialización a algún formato de archivo de un subconjunto de objetos seleccionado.
- Delegar en una base de datos para persistir mi modelo
    - Bases jerárquicas, en red, relacionales, NoSQL
    - Bases orientadas a objetos.

2.1 Snapshot del ambiente

Un snapshot es una foto de un momento determinado del ambiente (imagen). Algunos entornos como smalltalk ofrecen la posibilidad de guardar el mismo en un archivo para posteriormente recuperarlo. Esto permite tener una forma sencilla respaldar el estado de **todos** los objetos, incluso aquellos objetos del entorno no-estrictamente relacionados con la aplicación: herramientas de desarrollo, las ventanas de trabajo, las variables globales, etc.



Una de las grandes ventajas de esta solución es su simplicidad. El ambiente entero será persistido en un momento dado a un archivo físico. Generalmente los ambientes que soportan esta clase de serialización proveen las herramientas para realizar un snapshot de forma transparente. La estructura que tendrá ese archivo físico es binaria y estará muy intrínsecamente conectada a la representación real en memoria de los objetos (en última instancia un snapshot puede ser visto como un dump de la memoria del proceso, el cual puede luego ser recuperado).

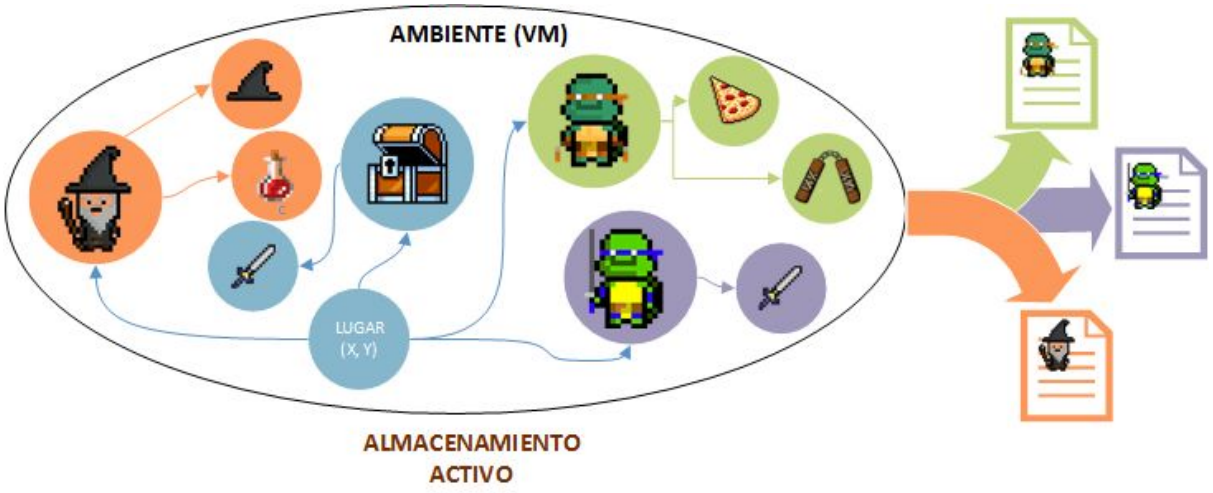
En la práctica esta solución resulta rara vez viable. Debido justamente a cada snapshot almacena todo el ambiente y no cambios incrementales. No resulta práctico generar un nuevo snapshot luego de cada operación que realicemos sobre nuestro sistema porque, como ya mencionamos antes, solo un muy bajo porcentaje de objetos sufre modificaciones a la vez. Adicionalmente, al trabajar con ambientes grandes un snapshot incurrirá en un costo significativo de performance.

Resulta útil sólo en aquellas aplicaciones monousuario en las cuales el estado no sea crítico y pueda ser guardado a intervalos discretos y grandes (cuando el usuario hace click en el botón guardar, al cerrar la aplicación, etc)

2.2 Serialización a archivos

Una alternativa para no guardar toda la imagen es seleccionar un subconjunto de objetos y escribirlos (serializarlos) a un archivo como un formato específico (a esta operación se la suele denominar “marshalling[1]”).

Una ventaja respecto al snapshot es que sólo se serializaran los objetos que nos interesen, con lo que el proceso de escritura y posterior lectura no solo será mucho más rápido porque incurrirá en mucho menos overhead. No obstante, debido justamente a que no se guardará todo el ambiente, tendremos que enfrentar la decisión de diseño de **qué** guardar, **dónde** y **cuándo**. Deberemos (de alguna u otra forma) seleccionar qué subgrafo dentro del grafo de objetos queremos almacenar en qué archivo



Es factible contar más de un archivo para más de un subgrafo. Una posible ventaja de esta solución es la siguiente: ante una operación que modifique solo uno de los subgrafos (por ejemplo, la agregación maguito + inventario) nos bastará con escribir nuevamente solo el archivo que almacena al maguito, sin tocar los archivos de “Leonardo” y “Miguel Ángel”.

También deberemos elegir el formato final de los archivos. El mismo puede ser variado. En todos los lenguajes existen distintas herramientas que nos ayudarán a serializar en los formatos más comunes.

2.2.1 Formatos binarios

Los archivos binarios suelen ser mucho más eficientes que los archivos de texto para almacenar objetos. Por lo general pueden ser leídos más rápido y ocupan sustancialmente menos espacio que los archivos de texto.

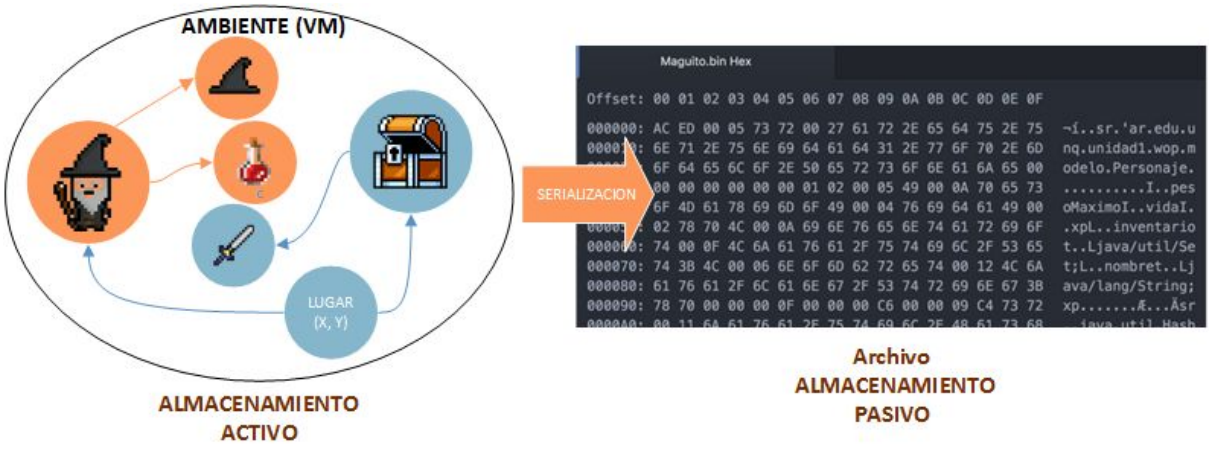
Ventajas:

- Más eficientes en tiempo lectura y escritura
- Más eficientes en espacio
- Por lo general tienen un mejor soporte para almacenar estructuras de tipo grafo.

[1] http://en.wikipedia.org/wiki/Marshalling_(computer_science),

Desventajas

- El resultado obtenido es “opaco” (no puede ser leído fácilmente)
- Los formatos suelen ser no estándares o basarse en estándares menos adoptados.
- En línea con lo anterior, muchas veces optimizan tiempo escribiendo en un formato muy cercano a la representación interna de cada objeto. Esto causa que la portabilidad de ese archivo sea más difícil.



Por ejemplo: si nuestro maguito tuviera un atributo de tipo Date (fecha de creación, por ejemplo) al persistirlo en un archivo binario es altamente probable que dicha fecha sea representado como un long (8 bytes) conteniendo la cantidad de milisegundos transcurridos desde el 01/01/1970 (epoch). Esto es así debido a que esa es justamente la representación interna en java del objeto Date. Al existir poca / nula conversiones de tipos entre lo almacenado y el objeto en memoria la serialización es sustancialmente más rápida, pero el archivo resultante estará por lo general mucho mas acoplado.

Algunas ejemplos de herramientas para escribir archivos binarios son:

- ObjectOutputStream (en java)
- ReferenceStream (en smalltalk)
- Apache Avro, Google protocol-buffer

2.2.2 Formatos de texto

Contrario a lo anterior, los archivos de texto suele priorizar la fácil lectura de los mismos por parte de un humano.

Ventajas:

- El resultado puede leerse (¡y editarse!) simplemente con un procesador de texto.
- Generalmente son formatos estándares muy bien definidos.
- Debido a lo anterior, es fácil portar dichos archivos de un sistema a otro.

Desventajas

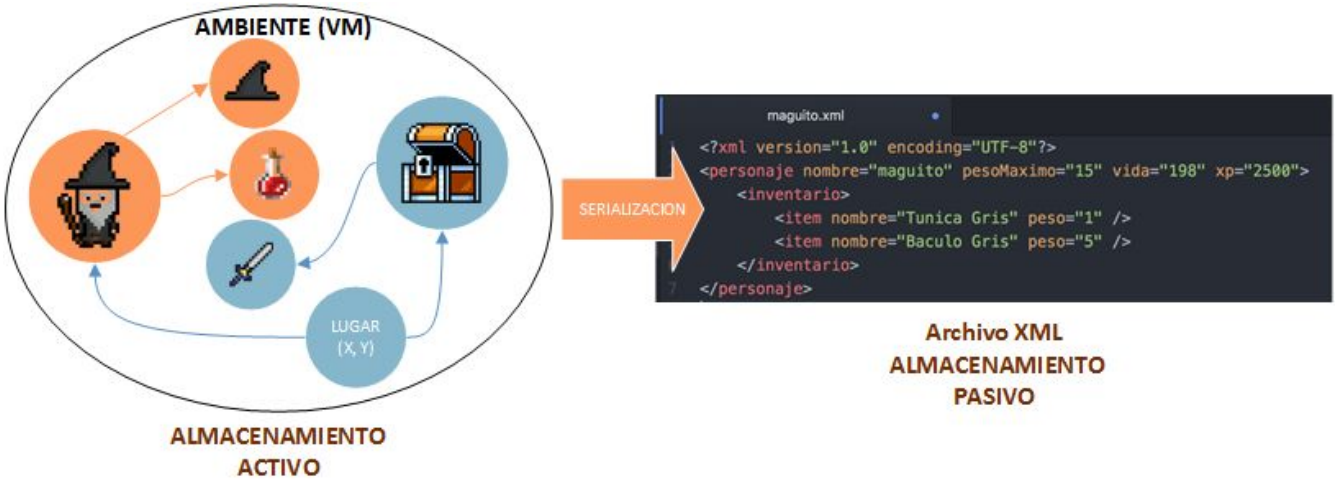
- Son menos eficientes: generalmente más grandes (alrededor del doble)
- Debido a que es necesario realizar muchas conversiones de tipo son más lentos para leer y escribir.

- Suelen ser limitados en cuanto a la estructura a persistir, por lo general solo soportan persistir árboles.

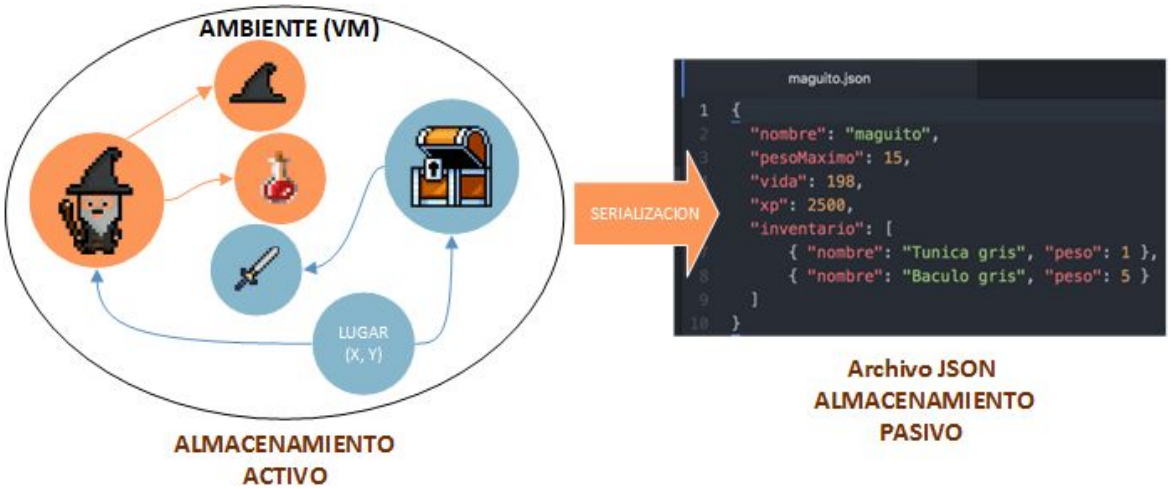
Una dificultad inherente a los formatos basados en texto (XML, JSON, YAML) es que están diseñados para persistir árboles. Un grafo de objetos NO es un árbol (aunque a veces puede asemejarse a uno en caso de no existir bucles). La herramienta que usemos para persistir tiene que poder “mapear” ambas estructuras de alguna manera (por ejemplo: insertando falsa hojas con referencias por id a otros nodos del árbol)

Algunos formatos comunes, con las herramientas que pueden usarse para persistir en ellos:

- **XML**: JAXB[2], XStream[3]



- **JSON**: FlexJSON[4], json-io[5], Google-gson[6], Apache Jackson



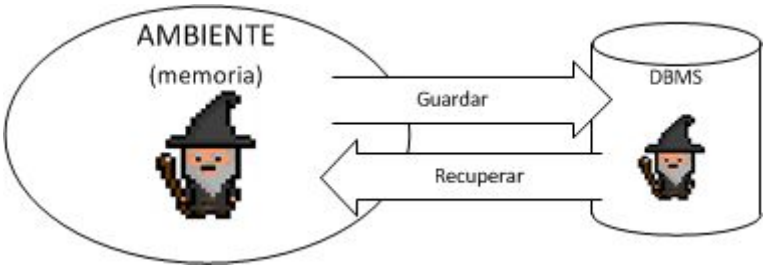
[2] https://jaxb.java.net/, http://es.wikipedia.org/wiki/JAXB, http://tech.deepumohan.com/2011/11/marshalling-in-java.html
[3] http://xstream.codehaus.org/, http://xstream.codehaus.org/tutorial.html
[4] http://flexjson.sourceforge.net/
[5] https://code.google.com/p/json-io/
[6] https://code.google.com/p/google-gson/

En el siguiente repositorio puede encontrarse un ejemplo muy sencillo de como persistir este modelo a binario, XML y JSON: https://github.com/EPERS-UNQ/unidad1-archivos

2.3 Servidores de base de datos

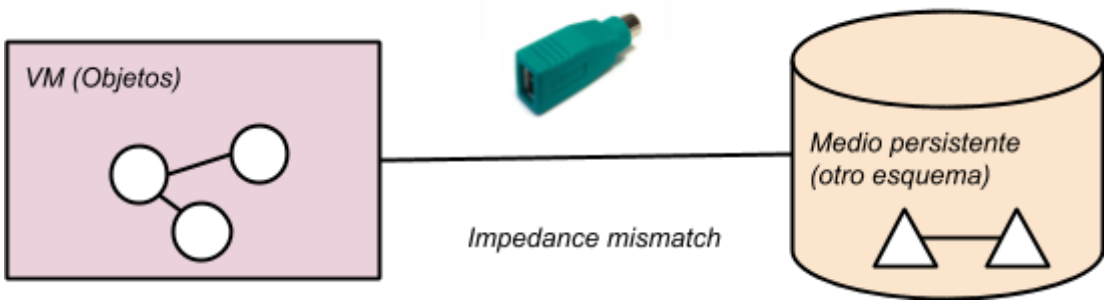
Otra solución posible es delegar en un servicio de base de datos el cual es una aplicación altamente especializada en persistir la información que le es suministrada.

Para ello utilizaremos típicamente el protocolo provisto por el mismo que nos permitirá tanto almacenar como consultar datos ya almacenados.



2.3.1 Bases de datos en otros esquemas

Típicamente los servidores de base de datos almacenarán la información en alguna estructura distinta a la de nuestro modelo de objetos. Para lograr la convivencia de ambos esquemas necesitaremos entonces hacer adaptaciones que cubran esas diferencias (lo que se llama **impedance mismatch**).



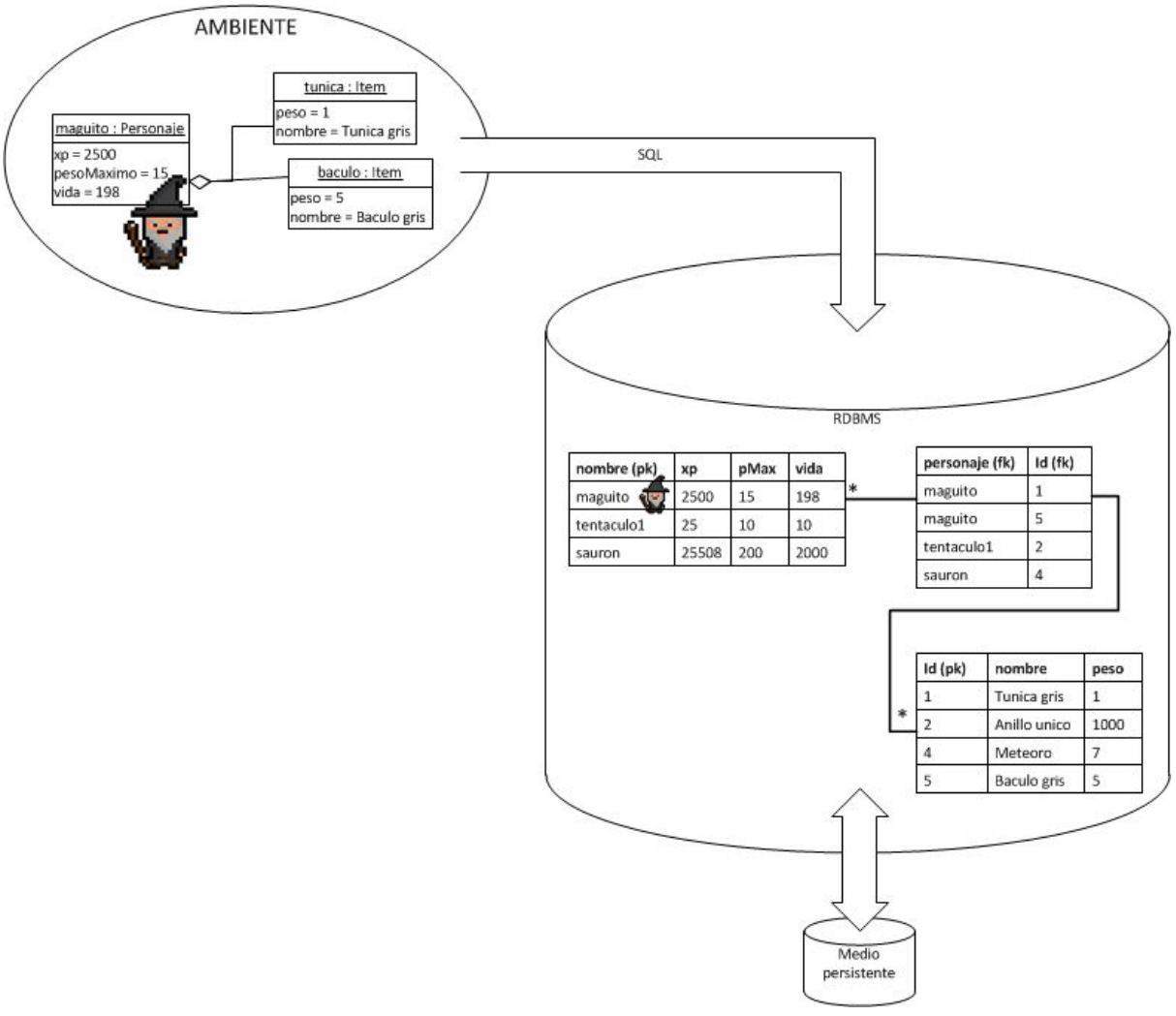
Algunas opciones posibles son:

- **Modelo relacional:** estructura la información en tablas. Cada tabla está compuesta de registros y cada registro tiene la misma cantidad de campos (columnas). Un registro en una tabla puede referenciar a otro registro en otra tabla por medio de una foreign-key. SQL (Structured Query language) es el estándar defacto para acceder a este tipo de bases de datos.
- **Modelo jerárquico:** se arma una estructura de nodos padre-hijo, donde cada *registro* hijo tiene un puntero físico a la estructura padre. Algunas implementaciones:
    - IMS de IBM (tenés un manual que tiene una intro bastante profunda, los capítulos *11 y 12* cuentan en detalle cómo es el acceso en una base jerárquica)
    - Natural Adabas de Computer Associates

- **Modelo en red:** es una variante mejorada de las bases jerárquicas donde no hay padres ni hijos, sino que se permite que cada estructura referencie a una o varias estructuras mediante *punteros*, además de trabajar con valores base.
    - podés ver un resumen en castellano del modelo CODASYL que definió el estándar de las bases de datos en red (disuelto en 1983), y también este artículo que explica ese modelo
- **Bases de datos NoSQL:** surgen como una alternativa a las bases de datos relacionales (SQL). Más adelante estudiaremos el trabajo con distintos NoSQL (grafos, clave-valor, documentos, etc).

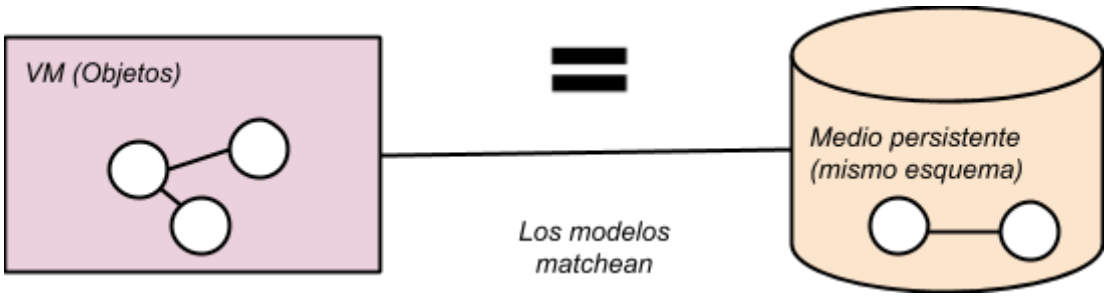
En todos los casos utilizaremos una API proviste por el servicio de base de datos para recuperar, insertar o modificar los datos almacenados en nuestras tablas. Al poseer ambas aplicaciones (nuestro ambiente y el servidor de base de datos) modelos distintos es tipico que una **unidad cohesiva** al pasar de un desde nuestro ambiente de objetos al modelo de datos se desmembrarse y se persista como varias entidades diferentes.

Esa adaptación dolorosa es lo que antes mencionamos como “impedance mismatch”



2.3.2 Bases de datos de objetos

Entre todas las variantes posibles podemos pensar que la opción más natural es tener el mismo esquema de objetos en la virtual machine y los que se almacenan en el medio persistente: **desaparece el impedance mismatch**.



Además tenemos en forma nativa

- Herencia
- Polimorfismo
- Referencias entre objetos.

La base de datos puede estar

- embebida dentro de un ambiente. Ejemplos: GemStone[7], Objectivity[8], y esquemas con prevalencia o similares.
- fuera del ambiente, corriendo como un proceso separado. Ejemplos: DB4O[9], ObjectDB[10], Realm[11], Magma, etc.[12]

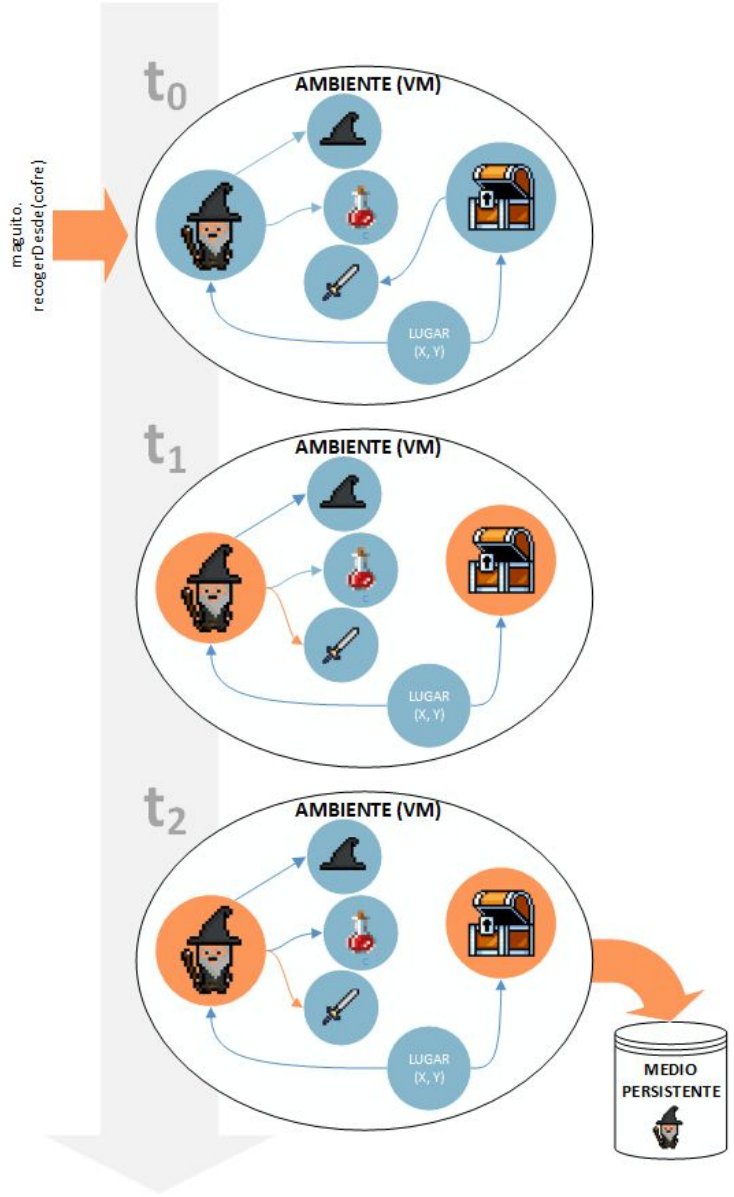
3 Estrategias de persistencia

Una vez elegido el medio donde guardar nos interesará analizar dos nuevas preguntas: ¿Cuándo guardar? y ¿Cómo mantener la consistencia? Existen varias estrategias para resolver esas preguntas.

3.1 Ambiente vivo y medio persistente como backup

[7] http://www.gemstone.com/
[8] http://objectivity.com/
[9] En 2014 Actian dejó de ofrecer DB4O como solución no-comercial para generar un producto pago (Versant Object Database)
[10] http://www.objectdb.com/tutorial/jpa/start
[11] https://realm.io/news/introducing-realm/, pensado para plataformas móviles como una alternativa a SQLite
[12] Puede verse una comparación de soluciones en http://en.wikipedia.org/wiki/Comparison_of_object_database_management_systems

Esta opción es a priori la menos invasiva a nuestro diseño. La idea principal es mantener todos los objetos vivos dentro de la imagen y operar sobre ellos de forma normal (por medio de un mensaje).



Cada cambio que suceda en el ambiente deberá luego ser sincronizado con el medio persistente. Dicha sincronización puede:

- ser sincrónica (no se terminará la operación hasta que los datos estén persistidos en el medio) o ser asincrónica (ocurrir en segundo plano).

- implicar guardar una foto de todo el ambiente completo luego de aplicarse algún cambio (snapshot), solamente el estado final de los objetos que se hayan modificado o incluso un journal de dichos cambios (prevalencia).

En esta solución es el modelo de objetos el dueño de los datos, el medio persistente se vuelve solo un respaldo. La aplicación solo leerá el medio persistente para recrear su estado en un momento cero, el resto de los acceso serán exclusivamente escrituras.

Los inconvenientes de esta solución son los siguientes:

- Al encontrarse todo el ambiente vivo esta estrategia no nos ayudará a sortear la limitación impuesta por el tamaño de la memoria.

- Deberemos asegurar que nuestro estado (en objetos) sea en todo momento consistente y que todo cambio al mismo sea atómico y durable. Volveremos sobre este punto en cuanto hablemos de transaccionalidad.

3.1.1 Esquema prevalente

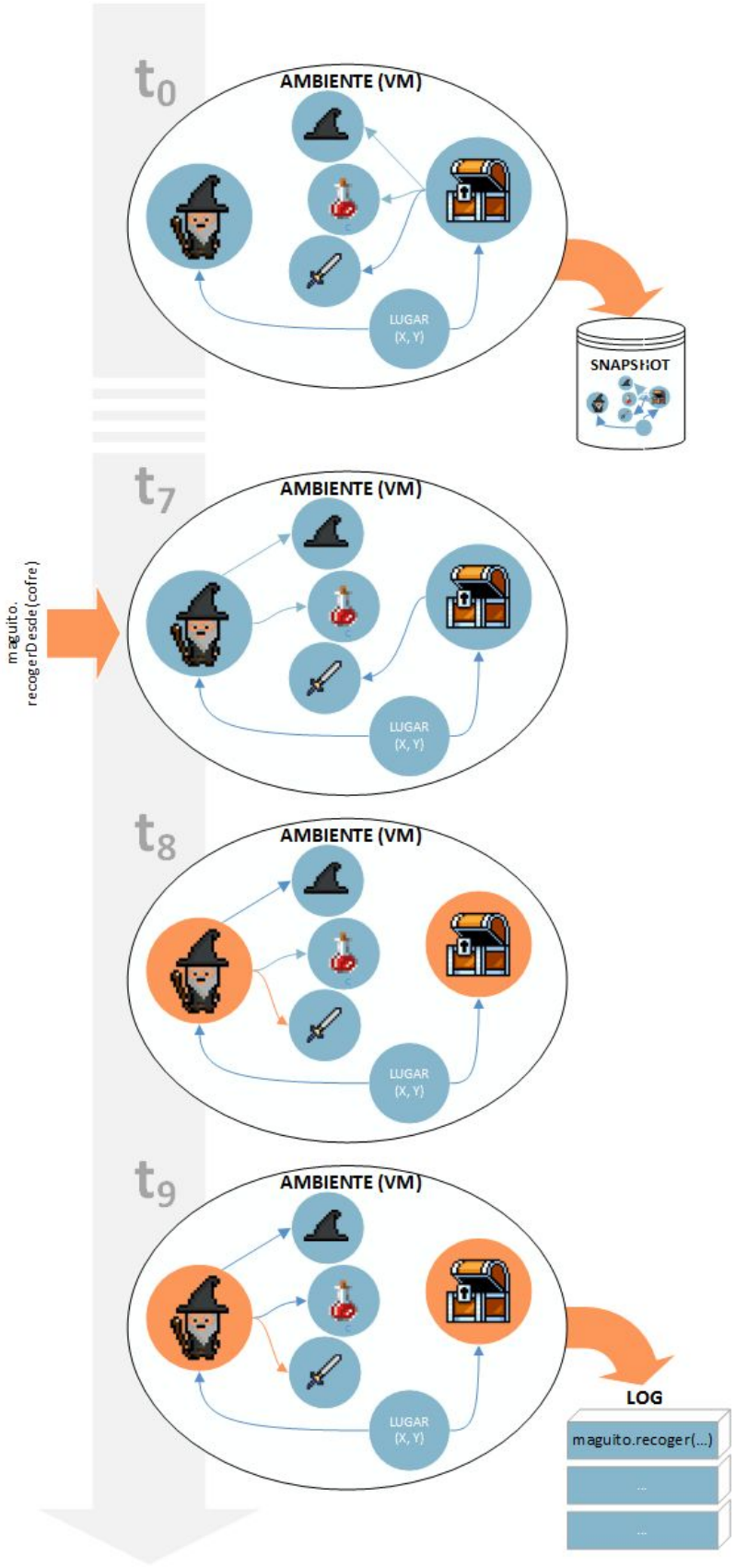
Como ya mencionamos, una variedad interesante del modelo anterior son los esquemas prevalentes. En los mismos todo el grafo de objetos se mantiene en memoria. Cada operación que afecte el estado es modelada como un objeto comando (patrón Command). En lugar de persistirse el resultado de cada operación lo que se hace es persistir la operación misma en un log (journal) de operaciones.

Eventualmente, a intervalos discretos bastante amplios (una vez al día, por ejemplo), se persiste un snapshot de todo del grafo de objetos y limpia el log de operaciones.

Cuando ocurre una falla en el sistema (corte de luz, error de hardware o bien falla en el software que cause la caída del servidor), se recuperan los datos a partir del snapshot + el reprocesamiento completo del archivo de log.

Características:

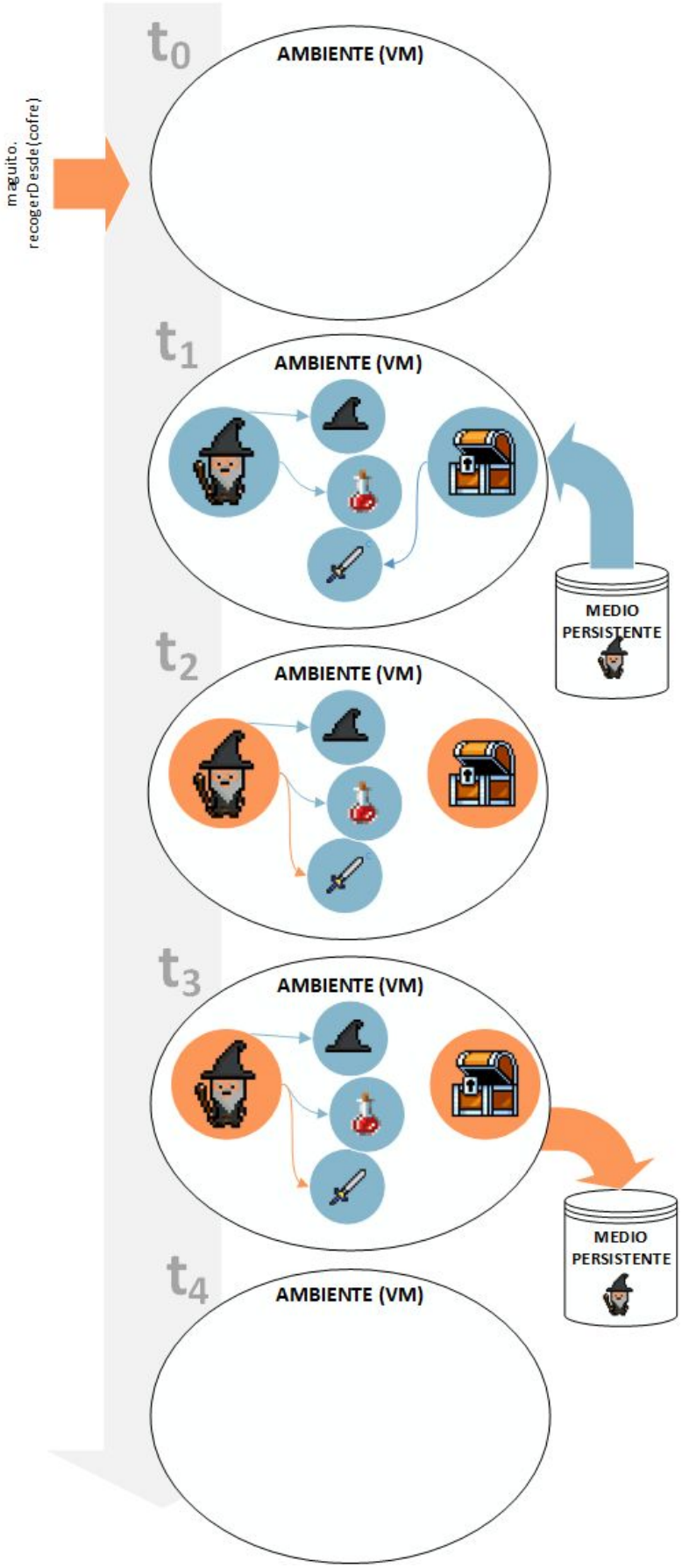
- es un esquema que soporta un muy alto número de transacciones por segundo (la memoria RAM tiene una tasa de transferencia notablemente superior al disco rígido).
- es relativamente simple de implementar, ya que sólo debemos indicar la manera en que se serializa cada objeto *y las operaciones que tienen efecto colateral sobre ellos (implementaciones de Command pattern).*
- requiere tener en memoria todos los objetos necesarios para correr la aplicación.



Para más información recomendamos echar un vistazo a la herramienta Prevayler[13].

[13] http://prevayler.org/

3.2 Ambiente muerto y recreado ante cada estímulo



Otra solución posible es la de mantener un ambiente vacío (o casi vacío). Cuando el mismo reciba un estímulo externo deberá **reconstruir su estado** (o el subconjunto del mismo que necesite para trabajar) desde el medio persistente, aplicar los cambios y luego persistir nuevamente dicho estado en el medio.

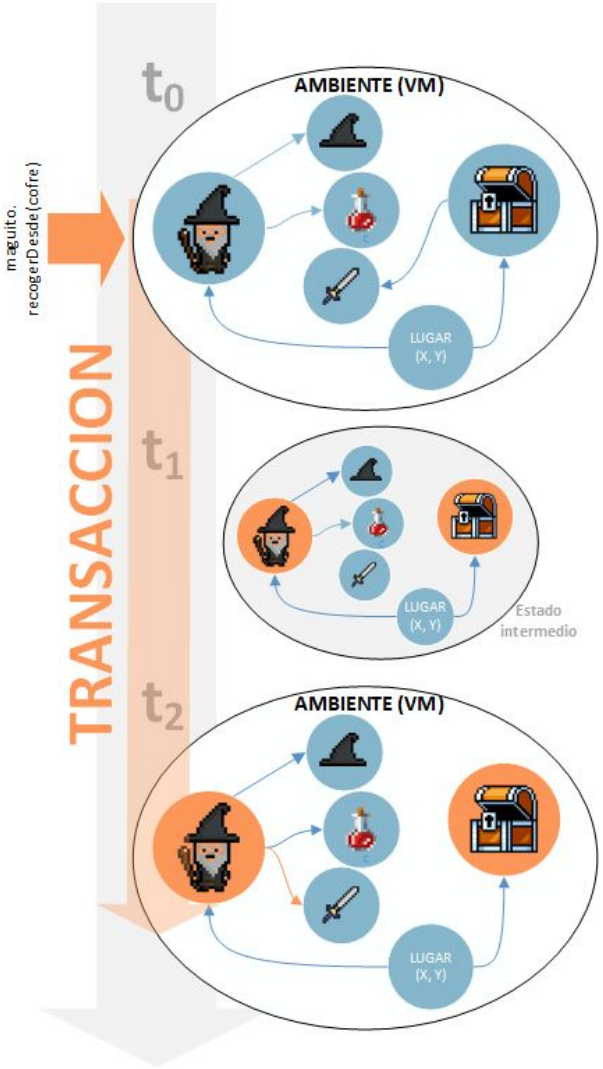
Las desventaja más notoria de esta aproximación es la **pérdida de identidad** de los objetos. En un esquema puro nuestro personaje maguito era modelado por exactamente una instancia la cual poseía una identidad fuerte. En este esquema nuestra instancia maguito es recreada varias veces (una vez por cada estímulo que el ambiente deba responder). Cada nueva instancia de maguito, si bien igual desde un punto de vista lógico, no será idéntica a la anterior (recordar *igualdad vs identidad*)

4. Transaccionalidad

Vamos a volver un poquito para atrás. Al principio del apunte definimos a una aplicación como la transformación continua de un cierto estado y definimos a las persistencia como el mecanismo para garantizar que dicho estado sobreviva.

Existe otra gran problemática relacionada con tener estado. Esta es lo que se denomina en inglés como *“reliability”* o la confianza que necesita tener una aplicación en que su estado es, en todo momento, correcto. Y de la mano esta necesidad es que surge el concepto de **transacción**.

Una transacción es un número de transformaciones al estado del sistema que suponen una unidad lógica (unit-of-work). Es decir, desde el punto de vista del negocio todos los pequeños pasos que integran una transacción deben ser vistos como una sola operación.



En nuestro juego el hecho de recoger un item desde un cofre y agregarlo al inventario el maguito es una transacción. Aunque lo hayamos estudiado hasta ahora como una unidad en realidad está compuesto de sucesivas pequeñas transformaciones del estado entre las cuales podemos enumerar:

- *remover el ítem de la colección del cofre*
- *agregarlo al inventario del personaje*

Para mi negocio las mismas deberían considerarse como una solo unidad de trabajo (unit-of-work). Los estados intermedios, si bien existen, no tienen ningún significado desde el punto de vista de negocio.

Una característica de toda transacción es que tienen un punto claro de inicio y de fin.

4.1 Propiedades ACID

En su concepción original (circa 1981) una transacción es definida como **un conjunto de operaciones sobre las que se aseguran (o deben asegurar) atomicidad, consistencia, aislamiento y durabilidad.** Las famosas propiedades ACID.

4.1.1 Atomicidad (Atomicity)

Todas las tareas que conforman una transacción son ejecutadas con éxito, o ninguna lo es. Este es el principio de “todo o nada”, si alguna de las transformaciones al estado no puede ser aplicada, ninguna es aplicada.

En ningún caso podrá ocurrir que el ítem desaparezca del cofre sin ser agregado al inventario del maguito, o que permanezca asociado tanto al cofre como al maguito.

Si durante la transacción surge alguna situación excepcional que no permite realizar alguna de las transformaciones intermedias (abortando así la transacción) todos los cambios ya hechos como parte de la misma deben ser deshechos.

4.1.2 Consistencia (Consistency)

Una transacción debe cumplir con todas las reglas definidas por el sistema (integridad de las relaciones, validaciones, restricciones, etc). El estado de la aplicación cumple con todas estas reglas al principio y al final de cada transacción y no existen a fines prácticos transacciones parcialmente completadas.

Supongamos que todo ítem tiene un peso en kilogramos. Existe una regla de negocio que nos dice que nuestro personaje no puede cargar más que cierto peso.

En ningún caso una transacción podría dejar a nuestro maguito cargando más kilogramos de los que él puede cargar.

4.1.3 Aislamiento (Isolation)

Este concepto fue el último en agregarse la definición de transacción (originalmente el acrónimo era C.A.D). Surge principalmente de la mano de la concurrencia (sistemas que responden a muchos estímulos al mismo tiempo, por ejemplo, muchos usuarios) y dice que el estado generado por una transacción no será visible por ninguna otra transacción hasta que ella no esté completada. Cada transacción es entonces independiente.

Supongamos ahora que el cofre puede ser visto por muchos jugadores. Si dos personajes intentan tomar el mismo ítem a la vez solo uno debería poder hacerlo (por más que las transacciones sean simultáneas deberían procesarse como si fuesen sucesivas).

En ningún caso ambos jugadores deben terminar con el ítem en su inventario.

4.1.4 Durabilidad (Durability)

Una vez que una transacción ha completado sus cambios persistirán en el sistema y no se descartaran.

Una vez concluida la transacción de recoger el ítem, el mismo no deberá desaparecer del inventario del jugador y volver al cofre por sí mismo.

4.2 Persistencia y transaccionalidad

Como ya vimos, **persistencia** y **transaccionalidad** son dos cuestiones inherentes a trabajar con estado. Si bien la transaccionalidad implica la persistencia (durabilidad), el inverso no es cierto. Podemos pensar en persistir sin ser transaccionales (sin cumplir todas las condiciones del modelo ACID). Es una decisión tecnológica va a estar estrictamente relacionada con el diseño de nuestro negocio.

Los servidores de bases de datos suelen proveernos de servicios no solo para persistir el estado sino que para garantizar que dicho estado sea manejado de forma transaccional. Es por eso que ambas problemáticas suelen atacarse en la práctica de forma conjunta.

En la práctica algunas de las propiedades de las transacciones suelen ser relajadas conforme a las implementaciones necesitan dar prioridad a otros requerimientos no funcionales: escalabilidad, distribución, performance, etc. El aislamiento es la característica donde es más fácil observar esto. La forma más sencilla de conseguir aislamiento es eliminando la concurrencia (ejecutando una transacción detrás de la otra en lugar de en paralelo) pero esto trae aparejado un impacto de performance significativo. Con el objetivo de lograr mitigar esto se proponen generalmente distintos “niveles” de aislamiento posibles. Los niveles más altos me aseguran una mejor exclusión mutua entre mis transacciones, los niveles más bajos me garantizan una mejor performance.

Lo importante es notar que **las características ACID no están escritas a fuego y que a veces contamos con la opción de vulnerarlas** conscientemente. La transaccionalidad es un requerimiento no funcional deseable en la mayoría de los sistemas, pero debemos tener en cuenta que viene a cambio de un costo. Desde la arquitectura es muy importante entender cuáles son las necesidades de cada negocio en particular.

En el caso de que mi juego fuese un juego online masivo (MMORPG). ¿Qué debería pasar si dos jugadores intentan tomar la espada del cofre exactamente al mismo tiempo? ¿Es necesario asegurar el aislamiento entre las dos transacciones? ¿O podríamos relajar esa condición? ¿Qué tan grave sería para mi negocio que ante una eventualidad dos jugadores terminen con la misma - o con dos copias de la misma - espada?

4.3 Transaccionalidad provista por el servidor de BD

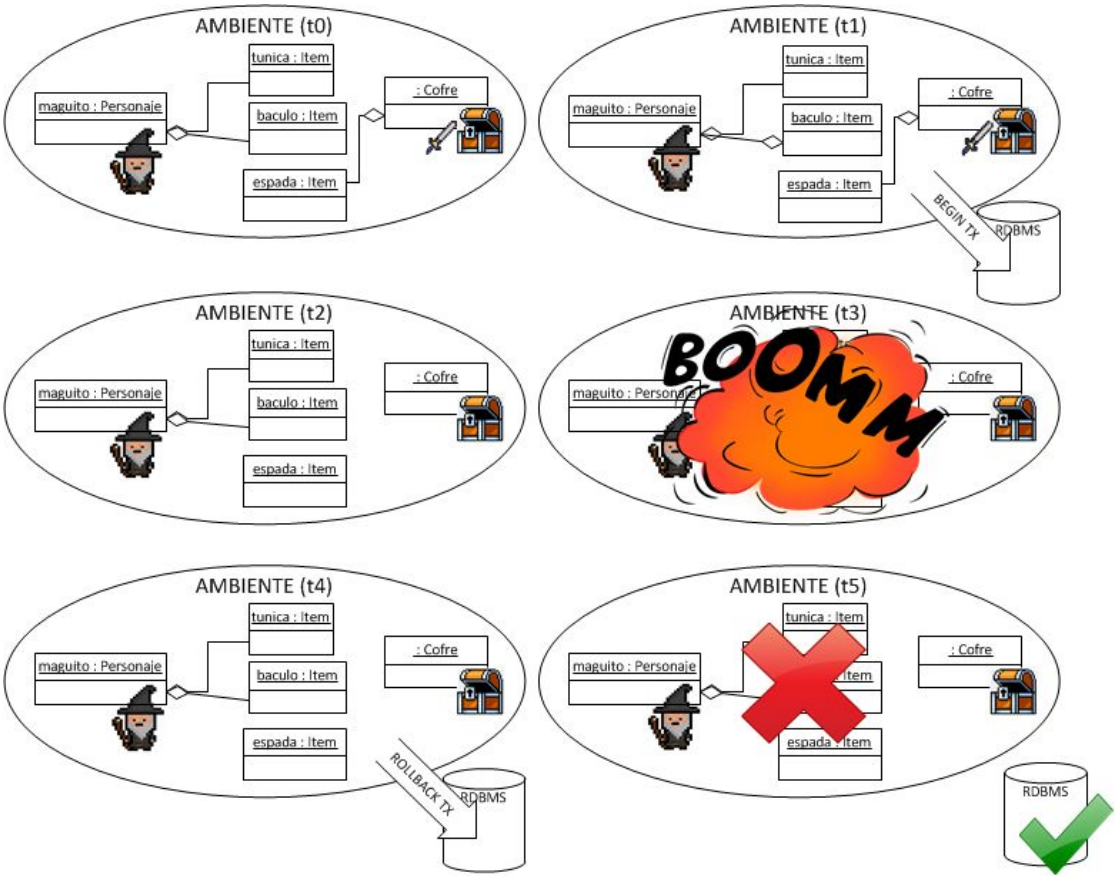
Como ya dijimos, es un feature común de los servidores de bases de datos contar con la capacidad de delimitar transacciones por medio de tres mensajes:

- Iniciar transacción
- Confirmar transaccion (commit)
- Abortar transaccion (rollback)

Utilizando estos mensajes, y basándonos en los dos estrategias de persistencia descritas anteriormente, tendremos los siguientes escenarios:

4.3.1 Ambiente vivo + transaccionalidad provista por BD

Si bien a priori este enfoque luce natural y sencillo tiene un problema fundamental: la transaccionalidad. **La base de datos solo nos puede asegurar transaccionalidad sobre el estado que ella misma contiene y no sobre los objetos que nosotros contenemos en nuestro ambiente.** Por lo tanto una situación excepcional en nuestro ambiente podría llevarnos rápidamente a una inconsistencia.

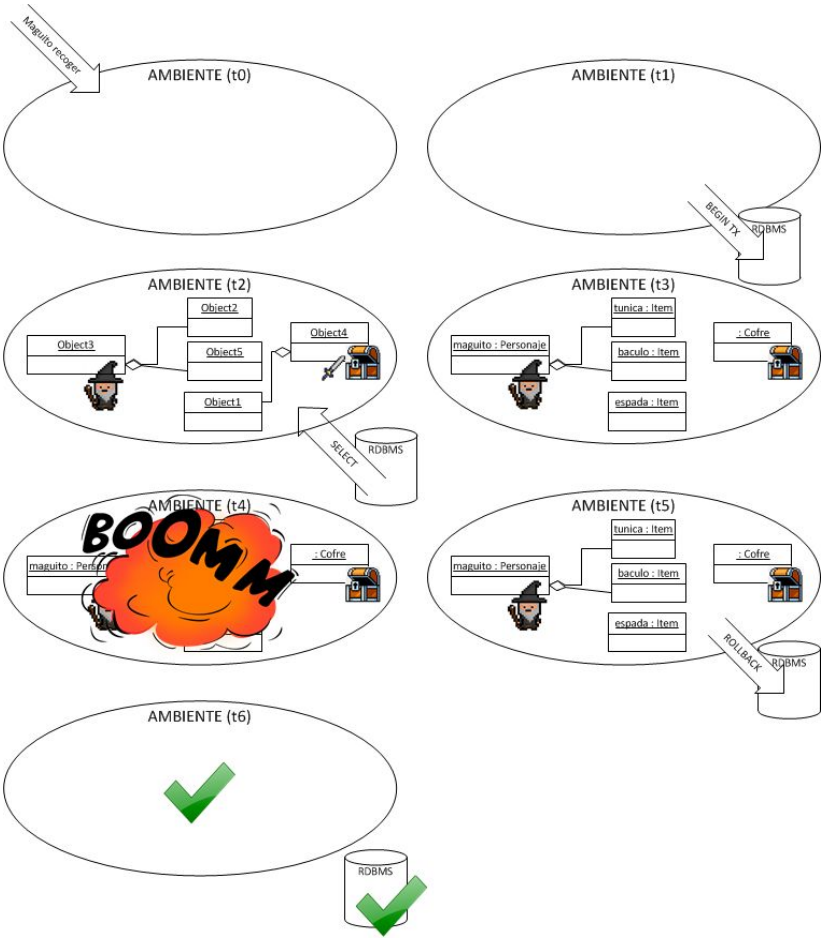


t_0 *- mi ambiente contiene al maguito con su inventario habitual. Se recibe un estímulo que nos ordena recoger la espada del cofre.*
t_1 *- se notifica a la base de datos el comienzo de la transacción*
t_2 *- se aplica una modificación parcial a mi estado*
t_3 *- alguna validación de negocio evita que mi operación pueda ser completada.*
t_4 *- se aborta (rollback) la transacción con la base de datos, todos los cambios en la base de datos son descartados.* ***No así los cambios parciales en nuestro ambiente*** *generados en* t_2. *Como consecuencia de esto la espada ha desaparecido.*

¿Es posible solucionar esta situación? Desde luego, podríamos también agregar la lógica para deshacer los cambios de parciales a nuestro ambiente y volver al estado original como existía en t_0, efectivamente implementado transaccionalidad sobre nuestro ambiente nosotros mismos.

4.3.2 Ambiente muerto + transaccionalidad provista por BD

En la solución es la de mantener un ambiente vacío se aprovecha enteramente la capacidad transaccional de la base de datos. Si alguna situación excepcional ocurriese que obligue a la transacción a ser abortada la base de datos quedaría en un estado consistente mientras que los cambios parcialmente en el ambiente sería descartados enteramente (porque siempre se descartan los objetos del ambiente en este modelo)



5 Resumen

Hemos presentado al lector una noción del concepto de persistencia como un mecanismo que permite recuperar el estado de un sistema, y su relación con otro concepto importante como es la transaccionalidad y el acceso concurrente a la información. Además hemos brindado un pantallazo de las diferentes opciones que puede manejar a la hora de diseñar la persistencia de una solución en objetos. En ella intervienen una gran cantidad de tecnologías posibles, donde el contexto ayudará al especialista de sistemas en el análisis de fortalezas y debilidades de cada opción.