Blog Zenika
IT Systems architecture, frameworks, and random thoughts

zenika
ARCHITECTURE INFORMATIQUE

14
sept.
2014

# Pattern Command: Undo, variations Compensation/Replay/Memento

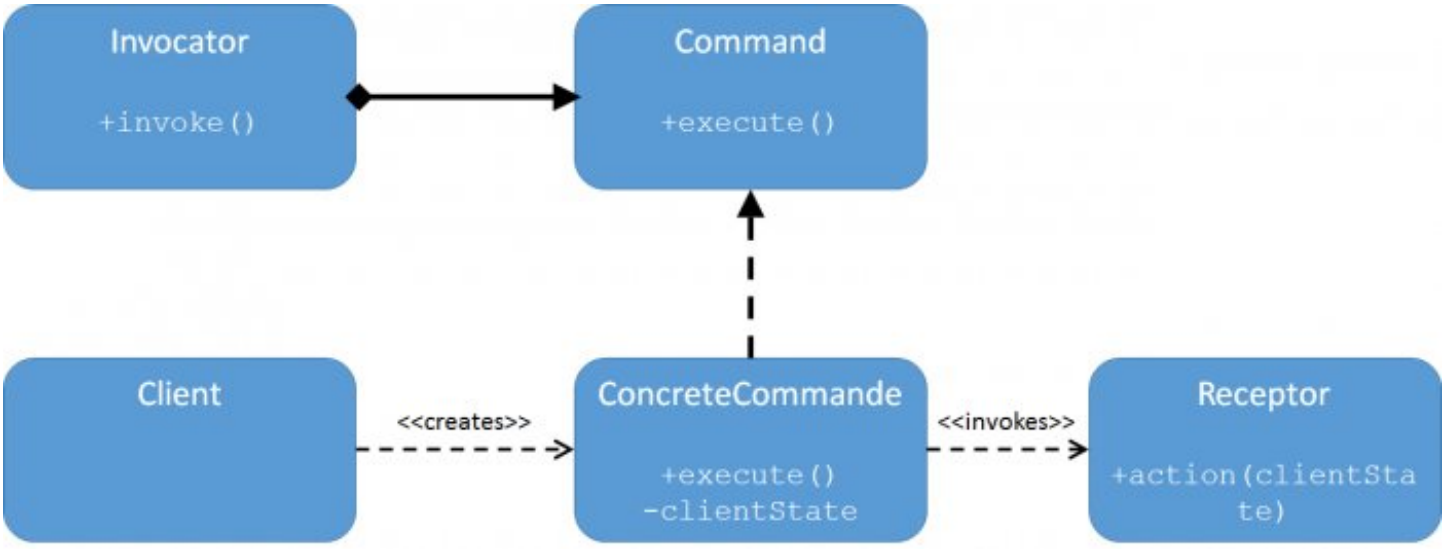
Posté par Laurent Claisse dans Développement › Software Craftmanship
Tags : java, patterns

La dernière fois je vous avez parlé de quelques patterns d'implémentation avec les enums java (http://blog.zenika.com/index.php?post/2014/06/11/Quelques-patterns-d-implementation-avec-les-enums-java) et en particulier de l'application des enums au Domain Driven Design grâce à l'inversion de dépendance. Continuons notre révision des bases avec le pattern Command; on va un peu plus loin en analysant en particulier 3 variations permettant d'implémenter l'undo/redo.

## Généralités

Selon le GOF, l'intention du pattern Command, est d'*encapsuler une requête comme un objet, autorisant ainsi le paramétrage des clients par différentes requêtes, file d'attente et récapitulatifs de requêtes, et de plus permettant la révision des opérations*. Le GOF fait ici allusion à deux utilisations des commandes: découpler la création d'une requête de son exécution, et implémenter l'undo. Ce dernier point est le sujet principal de ce post.

La structure est la suivante:



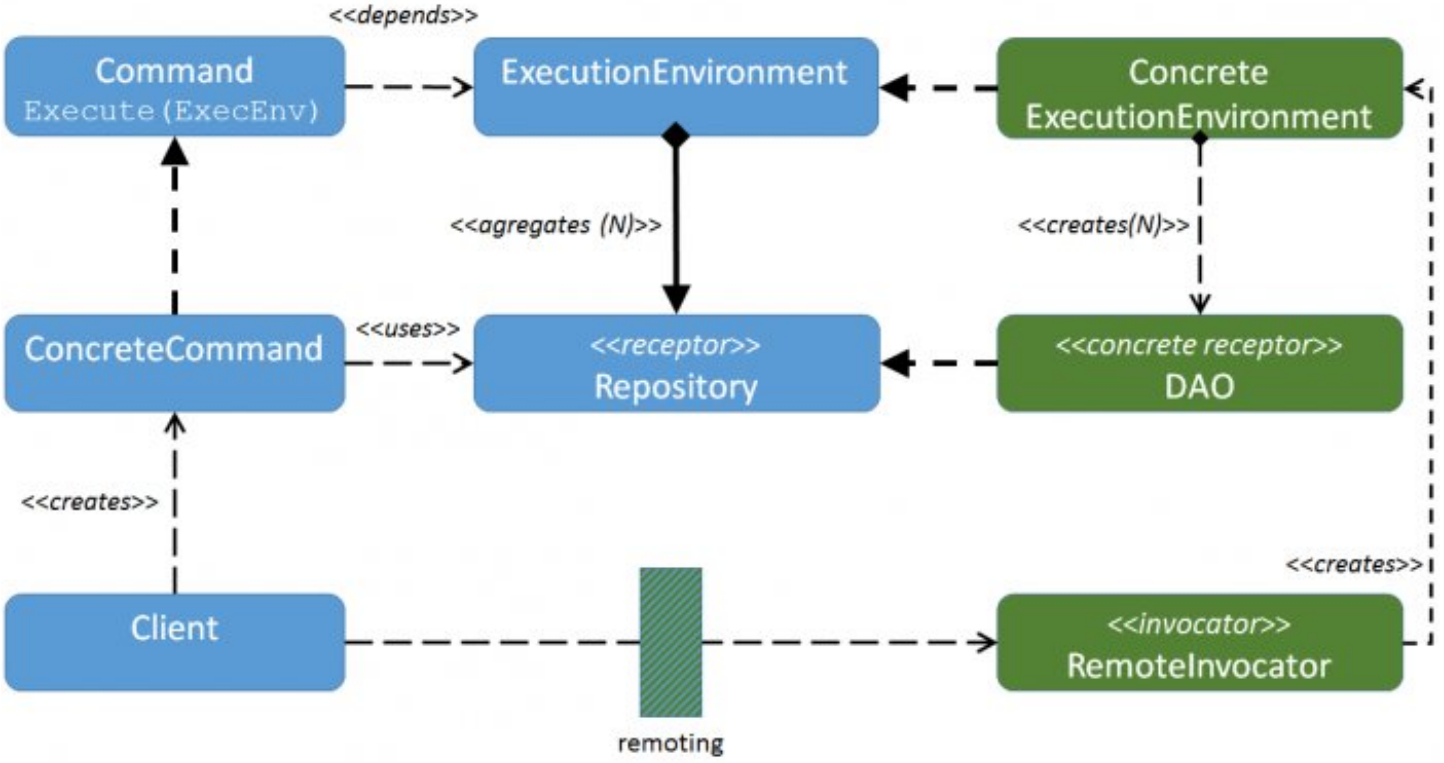

Lors de l'instanciation d'une ConcreteCommand par le Client, celle-ci capture l'état permettant de d'invoquer une action paramétrée. Lorsque l'Invocator déclenche une Command, la ConcreteCommand invoque le Receptor en passant l'état stocké en paramètre. Ce pattern permet de découpler l'instanciation d'un traitement de son exécution. On introduit souvent ce degré d'abstraction lorsque les lieux d'instanciation et d'exécution d'un traitement sont éloignés, dans l'espace ou dans le temps.

## Découplage spatial ou temporel avec le pattern Command

Le problème à résoudre est que le point de création de la commande, qui est le seul à connaître les données nécessaires à l'exécution d'un traitement, peut être différent du point d'exécution de la commande, qui est le seul à connaître le contexte(scope ou ressources techniques) nécessaires à son exécution.

### Découplage spatial: commande remote

Dans le cas spatial, la commande est typiquement créée dans un client et exécutée dans un serveur (les rôles du pattern Command sont indiqués sous forme de stéréotypes UML quand il diffèrent des types propres à la variation Remote Command):

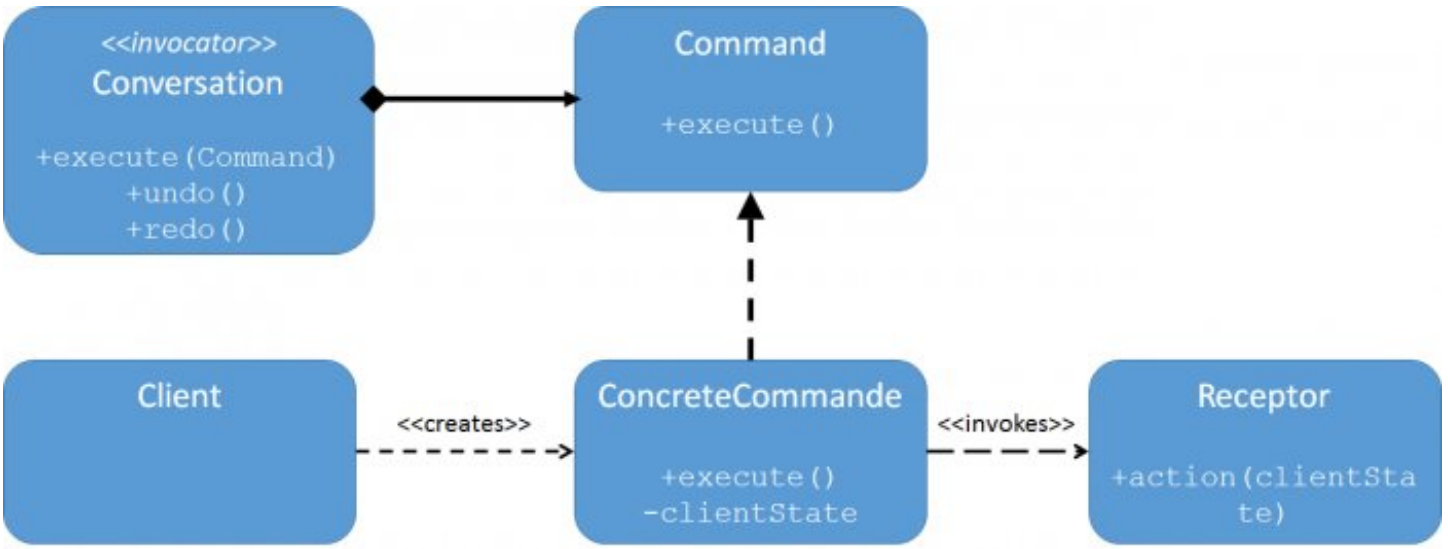



- Le constructeur de la commande concrète capture les données propres à une transaction client.
- La méthode execute de la commande concrète invoque un ou plusieurs récepteurs abstraits (ex: Repository).
- Pour ce faire, elle récupère les récepteurs abstraits dans l'environment d'exécution (abstrait lui aussi, ex: ExecutionEnvironment.getRepository)
- Côté serveur, la commande est reçue par le RemoteInvocator
- Les récepteurs concrets (ex: DAO implements Repository), et l'environnement d'exécution concret, sont implémentés côté serveur
- Le RemoteInvocator instancie un ConcreteExecutionEnvironment, en passant à son constructeur les ressources techniques nécessaires (EntityManager, ...). Le RemoteInvocator peut par exemple être un EJB @Remote.
- Le ConcreteExecutionEnvironment construit et agrège les récepteurs concrets (DAO, ...) en utilisant les ressources fournies par le RemoteInvocator (EntityManager, ...)
- Le RemoteInvocator invoque la méthode execute de la commande en passant le ConcreteExecutionEnvironment

**Découplage temporel: l'undo**

Dans le cas temporel, le problème est différent: la commande peut être exécutée tout de suite, mais elle doit pouvoir être annulée ou rejouée à un instant ultérieur et indéfini. Comme le dit le GOF, *un objet Commande peut avoir une durée de vie indépendante de la requête originelle*. Il est donc nécessaire qu'un contexte maintienne une référence vers la commande exécutée. Dans la suite on nomme ce contexte *Conversation*. La conversation mémorise les commandes exécutées et déclenche l'undo/redo. Ce type de commande est typiquement exécutée dans le même environnement que celui où la commande a été instanciée, et la méthode execute n'a donc pas besoin d'un paramétre ExecutionEnvironnement (les ressources nécessaires à l'exécution peuvent être passées dès l'instanciation).

L'undo par commande a la structure suivante (les rôles du pattern Command sont indiqués sous forme de stéréotypes UML quand il diffèrent des types propres à la variation Undo Command):




**Cas spatial et temporel---**

Il est évidemment possible de cumuler les deux difficultés, auquel cas le serveur devra à la fois fournir un ExecutionEnvironment, et maintenir une pile des commandes déjà exécutées (un EJB Stateful permet par exemple de remplir ces deux fonctionnalités). Puisque ce post concerne l'undo, on simplifiera cependant en supposant que les commandes sont locales (intra-JVM, sans remoting)

## L'undo

Venons-en au coeur du sujet: quelles sont les implémentation possibles? Laquelle choisir pour un type de commande particulier?

La fonctionnalité d'undo/redo est fréquemment demandée pour une IHM car l'être humain se trompe. Elle est souvent lié à une action utilisateur Ctrl+Z/Ctrl+Y. Elle n'est cependant pas triviale à implémenter, car les implémentations possibles dépendent du type de commande (on donne des exemples plus loin). On propose ici trois variations du pattern, plus ou moins adaptées selon le type de commande.

Le code est disponible sur Github (https://github.com/vandekeiser/cmd-pattern) . Il nécessite java 8. Le projet contient un framework d'undo par commandes. Les tests junit implémentent la commande *Typing* de saisie d'un message sur un affichage (*Display*). Il comprend une même suite de tests appliquée aux 3 variations.

**Les interfaces principales: Command et Conversation**

Puisqu'on se place dans le cas simple sans ExecutionEnvironment, l'interface Command est la suivante:

```
@FunctionalInterface
public interface Command {
    void execute();
}
```

L'annotation @FunctionalInterface n'est pas obligatoire, mais elle a les avantages suivants:

- Demande au compilateur de vérifier que notre interface est une Single Abstract Method Interface, qui peut être implémentée par un lambda
- Documente ce fait aux utilisateurs de l'interface, et signale l'engagement de l'auteur du framework à maintenir cette caractéristique dans les version futures

Introduisons la conversation, qui est le scope dans lequel on execute, annule, et rejoue des commandes:

```
public interface Conversation<C extends Command> {
    void exec(C cmd);
    void undo();
    void redo();
}
```

Le nom Conversation souligne le fait qu'il arrive souvent que le résultat d'une suite de commandes soit committé atomiquement, ce qui correspond à un scope conversation (on pourrait implémenter cette fonctionnalité avec Seam/CDI ou Spring Web Flow; ici on se contente d'instancier une conversation au début de chaque test). La conversation est identifiée à l'Invocator (du pattern Command du GOF), qui a le double rôle de mémoriser et de déclencher les commandes. Ici, c'est le même Client qui instancie les commandes concrètes et qui demande à l'Invocator de les déclencher. Le type parameter C est utilisé pour représenter les

spécificités des trois variations d'undo.

**Les tests exécutés pour toutes les variations**

Pour s'assurer que les 3 variations sont fonctionnellement équivalentes, on écrit les tests dans une classe abstraite CommandUndoTest_Typing. Les tests utilisent une commande TypeString qui représente la saisie d'un String dans un Display.

CommandUndoTest_Typing commence par les cas triviaux et va jusqu'à une conversation complexe:

```
/**
	 * Basic undo
	 * a    --> "a"
	 * undo --> ""
	 */
	@Test public void basicUndo() {
		Conversation<C> commands = newConversation();

		commands.exec(typeString("a"));
		assertEquals("a", display.displayed());

		commands.undo();
		assertEquals("", display.displayed());
	}

	/**
	 * Undo is noop when there were no execs
	 * undo --> ""
	 */
	@Test public void basicNoopUndo() {
		Conversation<C> commands = newConversation();

		commands.undo();
		assertEquals("", display.displayed());
	}

	/**
	 * Undo is noop when there were no execs, even when undo is called several times
	 * undo --> ""
	 * undo --> ""
	 */
	@Test public void basicNoopUndoTwice() {
		Conversation<C> commands = newConversation();

		commands.undo();
		assertEquals("", display.displayed());

		commands.undo();
		assertEquals("", display.displayed());
	}

	/**
	 * Undo is noop when there is nothing more to undo
	 * a    --> "a"
	 * undo --> ""
	 * undo --> ""
	 */
	@Test public void noopUndo() {
		Conversation<C> commands = newConversation();

		commands.exec(typeString("a"));
		assertEquals("a", display.displayed());

		commands.undo();
		assertEquals("", display.displayed());
```

```
        commands.undo();
        assertEquals("", display.displayed());
    }

    /**
     * Basic redo
     * a    --> "a"
     * undo --> ""
     * redo --> "a"
     */
    @Test public void basicRedo() {
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());

        commands.redo();
        assertEquals("a", display.displayed());
    }

    /**
     * Redo is noop when there were no undos
     * redo --> ""
     */
    @Test public void basicNoopRedo() {
        Conversation<C> commands = newConversation();

        commands.redo();
        assertEquals("", display.displayed());
    }

    /**
     * Redo is noop when there were no undos, even when redo is called several times
     * redo --> ""
     * redo --> ""
     */
    @Test public void basicNoopRedoTwice() {
        Conversation<C> commands = newConversation();

        commands.redo();
        assertEquals("", display.displayed());

        commands.redo();
        assertEquals("", display.displayed());
    }

    /**
     * Slightly more complex interaction between undo and redo
     * a    --> "a"
     * undo --> ""
     * redo --> "a"
     * undo --> ""
     */
    @Test public void exec_undo_redo_undo() {
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());
```

```
        commands.undo();
        assertEquals("", display.displayed());

        commands.redo();
        assertEquals("a", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());
    }

    /**
     * a    --> "a"
     * b    --> "ab"
     * undo --> "a"
     * undo --> ""
     */
    @Test public void typeA_typeB_undo_undo() {
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());

        commands.exec(typeString("b"));
        assertEquals("ab", display.displayed());

        commands.undo();
        assertEquals("a", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());
    }

    /**
     * a    --> "a"
     * b    --> "ab"
     * undo --> "a"
     * redo --> "ab"
     */
    @Test public void exec_exec_undo_redo() {
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());

        commands.exec(typeString("b"));
        assertEquals("ab", display.displayed());

        commands.undo();
        assertEquals("a", display.displayed());

        commands.redo();
        assertEquals("ab", display.displayed());
    }

    /**
     * a    --> "a"
     * b    --> "ab"
     * undo --> "a"
     * undo --> ""
     * redo --> "a"
     * redo --> "ab"
     */
    @Test public void exec_exec_undo_undo_redo_redo() {
```

```
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());

        commands.exec(typeString("b"));
        assertEquals("ab", display.displayed());

        commands.undo();
        assertEquals("a", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());

        commands.redo();
        assertEquals("a", display.displayed());

        commands.redo();
        assertEquals("ab", display.displayed());
    }

    @Test public void complexConversation() {
        Conversation<C> commands = newConversation();

        commands.exec(typeString("a"));
        assertEquals("a", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());

        commands.exec(typeString("b"));
        assertEquals("b", display.displayed());

        commands.exec(typeString("c"));
        assertEquals("bc", display.displayed());

        commands.exec(typeString("d"));
        assertEquals("bcd", display.displayed());

        commands.undo();
        assertEquals("bc", display.displayed());

        commands.redo();
        assertEquals("bcd", display.displayed());

        commands.undo();
        assertEquals("bc", display.displayed());

        commands.undo();
        assertEquals("b", display.displayed());

        commands.exec(typeString("e"));
        assertEquals("be", display.displayed());

        commands.exec(typeString("f"));
        assertEquals("bef", display.displayed());

        commands.undo();
        assertEquals("be", display.displayed());

        commands.undo();
        assertEquals("b", display.displayed());
```

```
        commands.undo();
        assertEquals("", display.displayed());

        commands.redo();
        assertEquals("b", display.displayed());

        commands.redo();
        assertEquals("be", display.displayed());

        commands.redo();
        assertEquals("bef", display.displayed());

        commands.redo();
        assertEquals("bef", display.displayed());

        commands.undo();
        assertEquals("be", display.displayed());

        commands.undo();
        assertEquals("b", display.displayed());

        commands.undo();
        assertEquals("", display.displayed());
    }
```

Les classes concrètes spécifiques aux 3 variations ne font qu'instancier un type différent de Conversation (à un détail technique propre aux génériques java près):

```
public class CommandUndoTest_Compensation_Typing_Test extends CommandUndoTest_Typing<CompensableCommand> {
    @Override protected Conversation<CompensableCommand> newConversation() {
        return new CompensationConversation();
    }

    @Override protected CompensableCommand typeString(String stringToType) {
        return new TypeString(display, stringToType);
    }
}

public class CommandUndoTest_Replay_Typing_Test extends CommandUndoTest_Typing<Command> {
    @Override protected Conversation<Command> newConversation() {
        return new ReplayConversation(()->{
            display.reset();
        });
    }

    @Override protected Command typeString(String stringToType) {
        return new TypeString(display, stringToType);
    }
}

public class CommandUndoTest_Memento_Typing_Test extends CommandUndoTest_Typing<MementoableCommand> {
    @Override protected Conversation<MementoableCommand> newConversation() {
        return new MementoConversation();
    }

    @Override protected MementoableCommand typeString(String stringToType) {
        return new TypeString(display, stringToType);
    }
}
```

**Coeur de l'implémentation**

Le coeur de l'implémentation est l'utilisation de 2 stacks (FIFO), une pour l'undo et une pour le redo: intuitivement, on sent bien que:

- chaque exécution de commande rajoute un undo potentiel
- chaque undo rajoute un redo potentiel

```
/**
 * @param <C> Command type
 * @param <S> Stack type
 */
public abstract class AbstractConversation<C extends Command, S> implements Conversation<C> {
	protected final Stack<S> undoStack, redoStack;

	public AbstractConversation() {
		this.undoStack = new Stack<S>();
		this.redoStack = new Stack<S>();
	}
}
```

On utilise un *type parameter* **S** en plus du type de commande **C**, car dans certaines variantes, on ne mémorise pas les commandes mais plutôt leurs effets.

La classe Stack est une classe interne au framework utilisée par les 3 variations:

```
public class Stack<T> {

	//Delegate to avoid exposing too many Deque methods
	private final Deque<T> stack = new LinkedList<>();

	/**
	 * @return null if stack is empty
	 */
	public T pop() {
		return stack.pollLast(); //Not using pop since it throws NoSuchElementException if the deque is empty
	}

	public void push(T cmd) {
		stack.addLast(cmd);
	}

	public void clear() {
		stack.clear();
	}

	public void forEachFifo(Consumer<? super T> action) {
		stack.stream().forEachOrdered(action);
	}
}
```

**Variation: Undo par Compensation**

L'infrastructure étant en place, regardons la première variation qui est la plus naturelle. Une action compensatoire d'une action A consiste à effectuer l'action inverse -A. Par exemple, l'action compensatoire d'un débit erronné de x EUR est un crédit de x EUR.

Dans le pattern Command, ceci se traduit par une commande compensable:

```
public interface CompensableCommand extends Command {
	void compensate();
}
```

On a vu dans le paragraphe *Découplage temporel: l'undo* que la conversation pouvait être identifiée à l'acteur *Invocator* du pattern Command du GOF. CompensationConversation est l'Invocator spécialisé pour les commandes compensables:

```
public class CompensationConversation extends AbstractConversation<CompensableCommand, CompensableCommand> {

	@Override public void exec(CompensableCommand todo) {
		todo.execute();
		undoStack.push(todo);
```

```
        redoStack.clear();
    }

    @Override public void undo() {
        CompensableCommand latestCmd = undoStack.pop();
        if(latestCmd==null) return;
        latestCmd.compensate();
        redoStack.push(latestCmd);
    }

    @Override public void redo() {
        CompensableCommand latestCmd = redoStack.pop();
        if(latestCmd==null) return;
        latestCmd.execute();
        undoStack.push(latestCmd);
    }
}
```

Dans la variation Compensation, les deux *type parameters* sont identiques (**S=C=CompensableCommand** dans **AbstractConversation<C,S>**), puisque l'état mémorisé est constitué de commandes. En effet, la compensation utilise les commandes déjà exécutées pour les compenser (undo) ou les réexécuter (redo).

A la fin d'exec(), on vide la stack de redo: les commandes undoées qui précèdent l'exécution d'une commande sont complètement effacées de la mémoire. La raison de ce choix est de se conformer aux conventions de toutes les IHM offrant un undo/redo (principe de moindre surprise).

**Variation: Undo par Replay**

Il est parfois difficile de trouver une action compensatrice. Dans ce cas si on a invoqué N commandes, une implémentation alternative de l'undo est de réinitialiser l'état à zéro, puis de rejouer les N-1 premières commandes. Le redo consiste ensuite à rejouer la Nième commande.

Contrairement à la variation *Compensation*, puisque la variation *Replay* se repose uniquement sur l'exécution des commandes dans leur sens normal, ReplayConversation n'a pas besoin d'un type de commande particulier de commandes: **S=C=Command** dans **AbstractConversation<C,S>**). Par contre elle a besoin d'une instance de commande capable de resetter l'état à zéro:

```
public class ReplayConversation extends AbstractConversation<Command, Command> {
    private final Command reset;

    public ReplayConversation(Command reset) {
        this.reset = reset;
    }

    @Override public void exec(Command todo) {
        todo.execute();
        undoStack.push(todo);
        redoStack.clear();
    }

    @Override public void undo() {
        Command latestCmd = undoStack.pop() ;
        if(latestCmd==null) return;
        redoStack.push(latestCmd);
        reset.execute();
        undoStack.forEachFifo(cmd->cmd.execute());
    }

    @Override public void redo() {
        Command latestCmd = redoStack.pop() ;
        if(latestCmd==null) return;
        latestCmd.execute();
        undoStack.push( latestCmd ) ;
    }
}
```

Le reset de l'état utilise une commande spécifique au type d'état modifié par les commandes. Dans le cas de la commande Typing (saisie) qui modifie un Display (affichage), on peut simplement effacer complètement l'affichage:

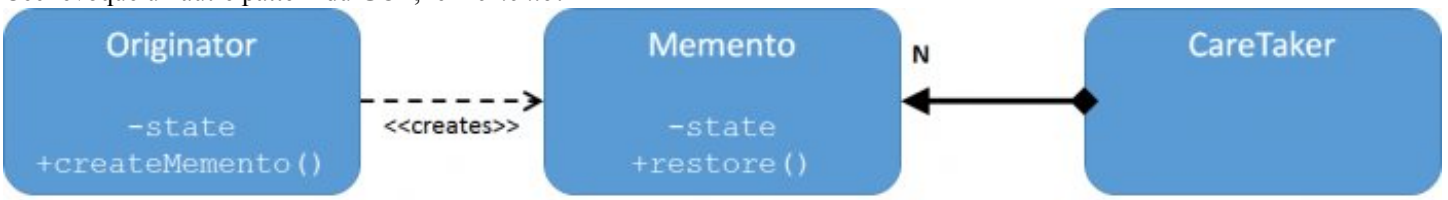
```
public class CommandUndoTest_Replay_Typing_Test extends CommandUndoTest_Typing<Command> {
	@Override protected Conversation<Command> newConversation() {
		return new ReplayConversation(()->{
			display.clear();
		});
	}
}
```

La commande **ReplayConversation.reset** est implémentée par le lambda anonyme qui vide l'affichage.

**Variation: Undo par Memento**

Plutôt que de mémoriser les commandes, on peut aussi implémenter l'undo en mémorisant leur effet, plus précisément l'état du système avant et après chaque exécution.

Ceci évoque un autre pattern du GOF, le *Memento*:




L'*Originator* instancie un *Memento* en passant à son constructeur un snapshot de l'état du système. Le *CareTaker* est chargé de connaître le Memento, afin de pouvoir demander, ultérieurement, la restauration de l'état capturé.

Le Memento doit être capable de restaurer un état capturé antérieurement, d'où l'exigence d'immutabilité. On ne veut pas voir les changements de l'état du système postérieurs au snapshot, il faut donc utiliser des techniques comme la *copie défensive*. Comme Memento est une interface, on précise cette exigence dans le contrat sous forme de javadoc:

```
/**
 * Implementations must be immutable (the memento must capture a snapshot)
 */
@FunctionalInterface
public interface Memento {
	void restore();
}
```

Chaque type de commande peut modifier un type d'état différent, pour lequel la façon de capturer un Memento sera différente. Par exemple la capture d'un état persisté en BD sera très différente de la capture de l'état d'un logiciel de dessin. Le plus simple est donc de spécialiser Command en MementoableCommand, pour que les implémentations de MementoableCommand réalisent à la fois la modification et la capture de cet état, suivant les spécificités de ce dernier:

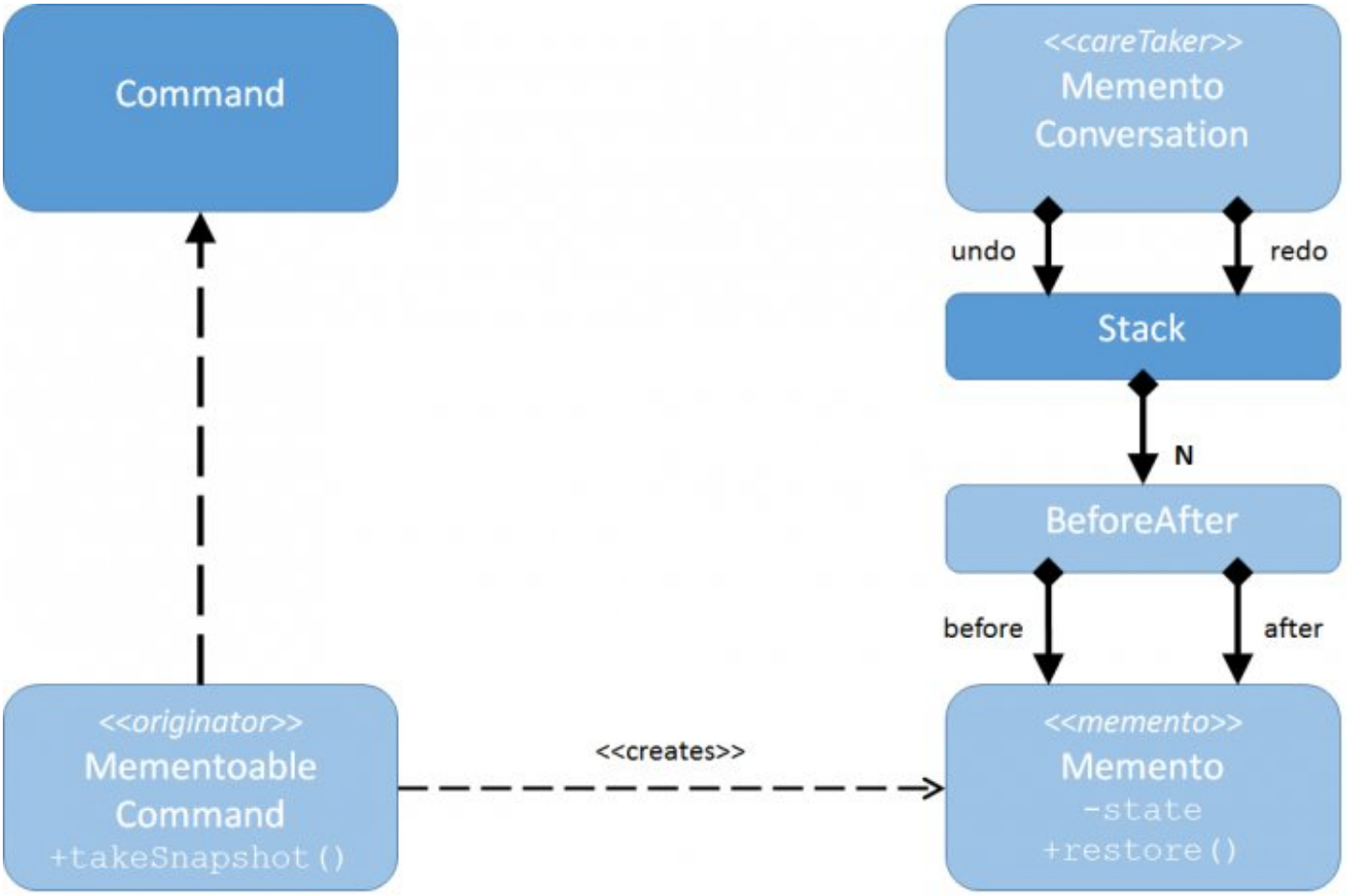
```
public interface MementoableCommand extends Command {
	Memento takeSnapshot();
}
```

Dans la variation Memento, les deux *type parameters* ne sont pas identiques (**C=MementoableCommand**, **S=BeforeAfter** dans **AbstractConversation<C,S>**), puisque l'état mémorisé est constitué de captures successives de l'état et non de commandes. Petite subtilité: pour implémenter le redo, on a besoin de capturer l'état avant ET après exécution de la commande:

```
public class MementoConversation extends AbstractConversation<MementoableCommand, BeforeAfter> {
```

```
    @Override public void exec(MementoableCommand todo) {
        Memento before = todo.takeSnapshot();
        todo.execute();
        Memento after = todo.takeSnapshot();

        undoStack.push(new BeforeAfter(before, after));
        redoStack.clear();
    }

    @Override public void undo() {
        BeforeAfter latestMemento = undoStack.pop();
        if(latestMemento==null) return;
        Memento latestBefore = latestMemento.before;
        latestBefore.restore();
        redoStack.push(latestMemento);
    }

    @Override public void redo() {
        BeforeAfter latestMemento = redoStack.pop();
        if(latestMemento==null) return;
        Memento latestAfter = latestMemento.after;
        latestAfter.restore();
        undoStack.push(latestMemento);
    }
}

//@Immutable
class BeforeAfter {

    final Memento before, after;

    /**
     * @param before must be immutable
     * @param after  must be immutable
     */
    public BeforeAfter(Memento before, Memento after) {
        this.before = before;
        this.after = after;
    }
}
```

Dans cette variation, la commande est donc l'*Originator* du pattern Memento du GOF (l'acteur qui déclenche le snapshot), et la conversation en est le *Caretaker* (l'acteur qui mémorise les mementos). Les rôles du pattern Memento sont indiqués sous forme de stéréotypes UML quand il diffèrent des types propres à la variation Memento Undo Command)




## Critères de choix d'une variation

Le critère éliminatoire est la possibilité d'implémenter une variation. Cette possibilité ou impossibilité dépend essentiellement de deux facteurs:

- Le type de *Receptor* des commandes: les tests communs aux 3 variations utilisent tous le même type de Receptor, la classe Display. Celle-ci implémente des méthodes permettant de choisir n'importe laquelle des variations. Ainsi unappend() est nécessaire pour implémenter la variation Compensation, getState()/setState() sont nécessaires pour implémenter la variation Memento; et clear() est nécessaire pour implémenter la variation Replay. Pour un Receptor réel, il est peu vraisemblable d'avoir autant de liberté de choix.
- Le type de commandes concrètes: certaines commandes ne se prêtent pas du tout à certaines variations.

Voyons pour chacune des trois variations quelques critères spécifiques.

**Compensation Undo**

Facteurs favorables:

- La Compensation est la plus naturelle (symétrie entre la commande et son inverse)
- Cette variation est idéale quand il existe une commande inverse naturelle dont le nom est évident: le contraire d'une création est une suppression (et vice-versa), le contraire d'un débit est un crédit (vice-versa)
- Ceci fonctionne bien quand la sémantique de la commande est très précise et a peu de degrés de libertés
- Du point de vue de l'utilisation mémoire, une stack de commandes est souvent plus lightweight que des snapshots de l'état complet (VS Memento),
- Du point de vue performance, elle évite de rejouer une séquence de commandes (VS Replay) ou de repositionner un gros état (VS Memento).

Facteurs défavorables:

- Les commandes non-compensables ne peuvent utiliser cette variation. Par exemple l'écriture dans un log ne peut être compensée.
- Une commande très générale comme un "update" est difficile à compenser: il faut mémoriser tous les champs mis à jour, utiliser l'introspection, ..
- Le nombre et la variété de commandes concrètes: pour choisir cette variation, il faut être sûr à l'avance qu'on pourra compenser toutes les commandes (même celles qui peuvent être introduites par une évolution future de l'application)

- Il est difficile de compenser les modifications de relations entre objets d'un graphe, surtout si ces relations sont bidirectionnelles

Remarques:

- La méthode Domain Driven Design préconise de toute façon d'éviter les commandes très générales comme l'update générique (POST), et de modéliser les changements d'état par des transitions déclenchées par des événements (**PUT /event {eventData}**). Les transitions ont une sémantique plus précise et sont plus facile à compenser.
- Quand tous les changements de l'état applicatif sont stockés comme une séquence d'événements (Event Sourcing), doit-on forcément utiliser les actions compensatoires? Pas forcément, tant que l'événement qu'on veut annuler n'est pas définitif (tant qu'il fait partie d'une conversation pas encore commitée).

**Variation: Memento Undo**

Facteurs favorables:

- Quand les commandes individuelles ne sont pas (ou difficilement) compensables, ou que le système présente une hysteresis
- Quand l'API du *Receptor* donne un moyen direct de réaliser des snapshots
- Quand on peut modifier ou wrapper un Receptor existant pour se ramener au cas précédent

Facteurs défavorables:

- Quand le Receptor est *write-only* ou qu'il est difficile de capturer son état
- Ne doit pas être rédhibitoire du point de vue des performances et de l'utilisation mémoire

**Variation: Replay Undo**

Facteurs favorables:

- *When all else fails*: cette variation est celle qui fait le moins d'hypothèses sur le Receptor: il suffit qu'il puisse être remis à zéro.
- Si on ne peut pas utiliser la variation Compensation, le Replay utilise dans la plupart des cas moins de mémoire car les commandes prennent moins de place que les mementos

Facteurs défavorables:

- Pas très élégant quand on peut faire autrement
- Ne doit pas être rédhibitoire du point de vue des performances

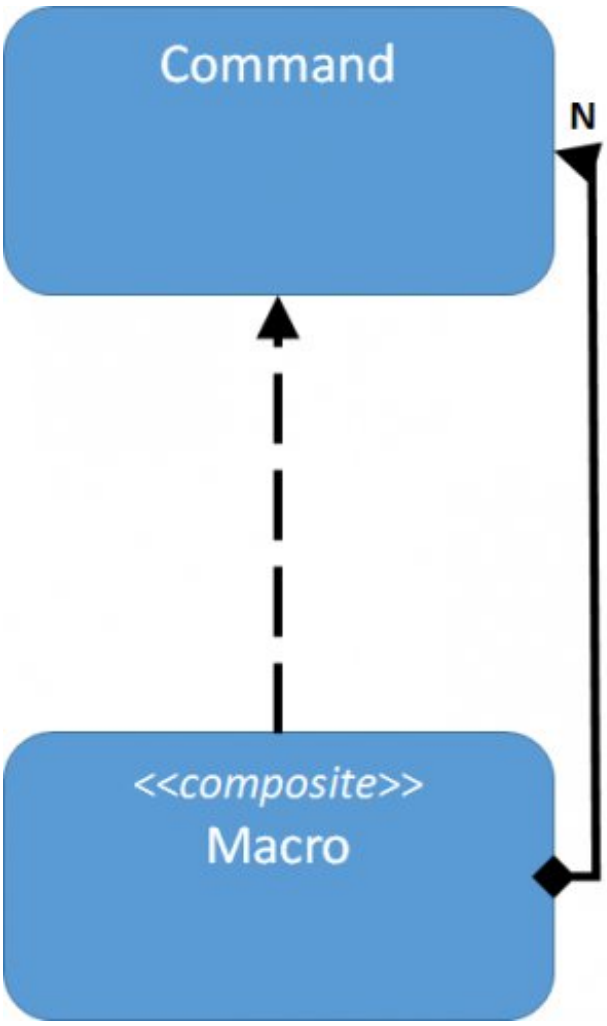
## Autres considérations

**Profondeur de l'undo-redo**

Dans notre exemple, les stacks d'undo/redo ont une profondeur illimitée. Ceci peut provoquer une OOME, et de toute façon l'utilisateur ne se rappelle plus des opérations trop anciennes. Le principe GRASP (attribution de responsabilités) *expert en information* nous suggère de placer la gestion de cette limitation dans le type qui a la connaissance de la profondeur actuelle de la stack, donc Stack.

**Commandes et macros**

A partir du pattern Command il est très simple de définir des macros: il suffit d'utiliser en plus le pattern Composite (les rôles du pattern Composite sont indiqués sous forme de stéréotypes UML quand il diffèrent des types propres à la variation Command Macro):




Fil des commentaires de ce billet

## Rechercher

ok

## A la une / A venir

- Introducing Spring MVC test framework
- Using Tomcat JDBC connection pool in a standalone environment
- AngularJS : la philosophie
- Intégrer ElasticSearch dans une application Java
- Creating a Varnish 4 module

## Catégories

- Annonces Zenika (41)
- Architecture (17)
  - Cloud et Datagrids (21)
- Développement (88)
  - Web et Mobile (51)
  - Autres Langages (5)
  - Software Craftmanship (2)
- Tests et Performance (8)
- Usine logicielle (33)
- Reporting (8)
- Formations (10)
- Evénements (142)

- Méthode Agile (1)
  - Scrum (1)
- DevOps (1)

## Tags

- java
- spring
- gradle
- eclipse
- html5
- birt
- gwt
- agile
- android
- javascript

**Tous les mot-clés**

## Formations

- Java Specialist
- AngularJS
- Management 3.0
- Continuous Integration
- Découverte de l'Agilité
- BPEL
- Wicket
- Toutes nos formations...

## Blogs

- The Coder's Breakfast
- Grégory Boissinot
- Nayima

## Partenaires

- Actuate
- ej-technologies
- Nayima
- SpringSource
- Terracotta

## Archives

- Accueil -
- Archives

---

Copyright Zenika, tous droits réservés. | Icônes FamFamFam