目的・範囲: *Objective & Scope*	Design Pattern(Mamento パターン)
分類名: *Classification Name*	
著作者: *Author*	沼尾　英
実施日: *Enforcement day*	2014年9月13日
バージョン: *Version*	
初版発行日: *Original Release*	2014年9月13日
現行版発行日: *Current Release*	2014年9月13日
キーワード: *Key Words*	
背景情報: *Background*	

◇目的

メメントパターンを使用する目的は、一度変化してしまったオブジェクトを「少し前の状態に戻したい」「ある時点の状態に戻したい」場合に
使われます。
テキストエディタ等で実装されているような「アンドゥ」(操作をキャンセルして操作前の状態に戻す)機能を提供するためのパターンです。

◇効果

オブジェクトの任意の時点の状態のクローンを作成し、保存しておくことで、その時のオブジェクトの状態を復元することを可能にします。

◇背景

プログラム実行中に、オブジェクトの状態がどんどん変化することが考えられます。一度変化してしまったオブジェクトを、「少し前の状態
に戻したい」「ある時点の状態に戻したい」などの要求は時に発生するものです。このような要求に応えるために設計されたのがメメントパ
ターンになります。

◇メメントパターンの実際のコードと考え方

Originator：状態を記憶すべきであるオブジェクト。
Memento ：内部状態を保持するオブジェクト。Originatorから生成され、Originatorオブジェクトの内部状態を保持します。
Caretaker：内部状態を管理するオブジェクト。Originatorオブジェクトの状態を保存／復元する役割を果たします。

以下サンプル

「Originator」の内部情報(フィールド値)を保持するクラス

```
// 「Originator」の内部情報(フィールド値)を保持するクラス
public class Memento {
   int param1;
   String param2;
   Memento(int param1, String param2) {
      this.param1 = param1;
      this.param2 = param2;
   }
   public int getParam1() {
      return this.param1;
   }
   public String getParam2() {
      return this.param2;
   }
}
```

自身の状態を「Memento」として保持し、与えられた「Memento」から自身の状態を復元するクラス

```
// 自身の状態を「Memento」として保持する
// 与えられた「Memento」から自身の状態を復元するクラス
public class Originator {
   private int param1;
   private String param2;
   public Originator() {
      this.param1 = 0;
      this.param2 = "";
   }
   public void countParam1(int param1) {
      this.param1 = this.param1 + param1;
   }
   public void appendParam2(String param2) {
```

```
    public void appendParam2(String param2) {
        this.param2 = this.param2 + param2;
    }
    public int getParam1() {
        return this.param1;
    }
    public String getParam2() {
        return this.param2;
    }
    /// 自分の状態を保存した「Memento」を作成するメソッド
    public Memento createMemento() {
        return new Memento(this.param1, this.param2);
    }
    // 要求された「Memento」に状態を戻すメソッド
    public void restoreMemento(Memento memento) {
        this.param1 = memento.param1;
        this.param2 = memento.param2;
    }
}
```

「Originator」の状態を保存したい場合、「Memento」を保持する
「Originator」の状態を元に戻したい場合、「Memento」を受け渡す

```
public class Caretaker {
    Memento memento;

    public Memento getMemento() {
        return this.memento;
    }

    public void setMemento(Memento memento) {
        this.memento = memento;
    }
}
```

利用者（メイン）

```
public class Client {
    public static void main(String[] args) {
        // オブジェクトを生成する
        Originator original = new Originator();
        // オブジェクトの状態を設定する
        original.countParam1(1);
        original.appendParam2("aaa");
        System.out.println("Original value : " + original.getParam1() + ", " + original.getParam2() + ".");

        // Mementoを取る（今の状態を記憶する）
        Memento snapshot = original.createMemento();
        // CaretakerでMementoオブジェクトを保持する
        Caretaker caretaker = new Caretaker();
        caretaker.setMemento(snapshot);

        // その他操作
        original.countParam1(3);
        original.countParam1(4);
        original.appendParam2("bbb");
        original.appendParam2("ccc");
        System.out.println("Updated value : " + original.getParam1() + ", " + original.getParam2() + ".");


        // Mementoによって復元する
        original.restoreMemento(caretaker.getMemento());
        System.out.println("Restored value : " + original.getParam1() + ", " + original.getParam2() + ".");
    }
}
```

出力結果

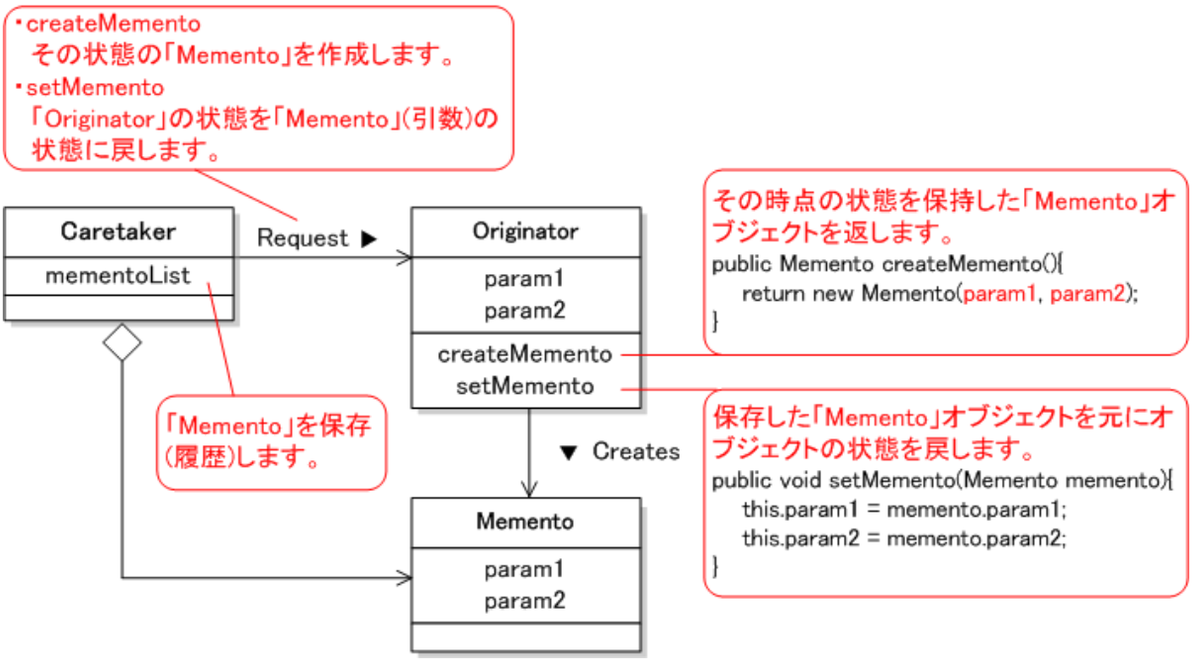
```
Original value : 1, aaa.
```

```
Updated value : 8, aaabbbccc.
Restored value : 1, aaa.
```

◇問題点の改善

Mementoオブジェクトは、Originatorのデータを保存しておかなければなりません。ということは、Originatorが複雑になるほど、その保存
作業が大変になることがわかります。例えば、Originatorが可変的なビットマップを保持している場合、Mementoオブジェクトを生成するた
びに、非常に大きなメモリのコピーが必要になります。これでは時間もかかりますし、メモリを圧迫します。
そのような場合、差分データだけを保持する形式にすると、大幅にメモリと時間の節約になることがあります。ビットマップも10ピクセル程
度しか変更していないのならば、その変更箇所だけを保存しておけば、大きなメモリを占有することもなくなります。

◇メメントパターンのまとめ



◇注意点

メメントパターンを用いる場合、originatorがほかのオブジェクトやリソースを変更してしまうかどうか注意が必要です。メメントパターンは
単一のオブジェクトに対して働くためです。

◇総括

実装するとすごく便利です。