# Preparación

* En un blog de notas, colocar la dirección para que descarguen el código.

<https://github.com/snahider/Head-First-Design-Patterns> (clone o download)

* Colocar las direcciones para que descarguen Resharper, en el caso de VS2010.

<http://www.jetbrains.com/resharper/>

(Opcional) Colocar las direcciones para que descarguen <http://update.improvingworks.com> (Eclipse: Help, Software Updates)

<https://github.com/continuoustests/AutoTest.Net/downloads>

# Introducción

El objetivo de todo esto, del Clean Code, de lo que vamos a ver Refactoring de tantas otras técnicas y prácticas, tienen que ver con que nosotros podamos entenderlos

Ya que usualmente pasamos más tiempo leyendo el código que escribiéndolo y por eso es muy importante la legibilidad, mantenibilidad, etc ya que ahorran bastante tiempo y dinero.

# Qué es Refactoring

Es modificar el código para poder alterar su estructura interna de tal manera que podamos mejorar algún atributo de calidad sin que el funcionamiento de la aplicación se ve a fectado.

Es algo interno relacionado al código y no a la funcionalidad, es decir refactorizar no es agregar nueva funcionalidad.

Es una técnica disciplinada, no es cambiar código por cualquier lado.

# Porqué es importante

Porque necesito yo modificar algo que no está roto, si supuestamente está bien.

Muchos de las veces pueden ser preventivos (reducir bugs, aumentar la reusabilidad y mantenibilidad), u otros estratégicos (nueva funcionalidad

# State Pattern

## Explicar el Dominio

Utilizando el gráfico de la página: 402 (428 / 681)

Máquina Dispensadora de Goma de Mascar que atravesará por múltiples estados dependiendo de la acción que nosotros ejecutemos sobre la máquina.

Explicar los estados y acciones

## Explicar el código

* En la parte superior se encuentran los estados por los cuales puede atravesar la máquina y su estado inicial será el de SOLD\_OUT;
* Cada método representa una de las acciones que se pueden realizar sobre la máquina: Insertar Moneda, Girar Manivela, Expulsar Moneda, Retornar Goma.
* Cada condicional dentro del método representa al estado en el cuál se puede encontrar la máquina al momento de ejecutar la acción y como reaccionará esta.
* Las reacciones que vamos a observar que realiza la máquina son: devolver mensajes de éxito o error al usuario, cambiar de estado.
* Acompañando al código, se encuentran los Test Unitarios para la Máquina. Antes de continuar con cualquier cosa vamos a EJECUTAR LOS TESTS.

## ¿Cuál es el Problema?

* Si queremos agregar un nuevo estado, ejem: Winner State. Vamos a tener que modificar cada método para agregar un nuevo condicional. Lo cuál es poco mantenible.
* No sigue el Open Closed Principle
* No hay encapsulación.

## Refactorización Propuesta

Utilizar las páginas del libro 411 (437 / 681).

El libro nos sugiere realizar la siguiente refactorización:

* El comportamiento de cada estado, lo encapsulemos en su propia clase. De tal manera que si tenemos que modificar un estado solo lo hagamos en la clase correspondiente a ese estado y no por todo lado. Entonces luego de realizar este cambio tendríamos una clase para el Sold, NoQuarter, etc.
* Cuando se realice alguna acción sobre la máquina, simplemente esta delega a su clase actual, como se va a comportar frente a la acción.

Los pasos que nos sugiere el libro, son los siguientes:

* Crear una interface State y agregar un método por cada acción que se puede realizar en la máquina.
* Implementar una clase State por cada estado en los cuales puede estar la máquina. Estas clases serán las responsables de realizar el comportamiento de la máquina, cuando esta se encuentre en ese estado.
* Eliminar todos nuestros condicionales, moverlos a la clase estado y delegar al estado el comportamiento de la máquina.

# State Pattern

No es otra cosa que un patrón que nos permite alterar el comportamiento de un objeto cuando cambiamos su estado interno.

En el gráfico de este patrón observamos:

* Existe una clase “Context” que es la que tendrá diversos estados internos. Ejm: Gumballmachine.
* Cada vez que se realiza el llamado a esta clase, se termina delegando la responsabilidad a alguno de sus estados.
* Definimos una interface para representar a todos las clases concretas de los estados, los estados deberán implementar esta interface de tal manera que puedan ser fácilmente intercambiados.

## Refactorización

* *Podríamos empezar creando la interfaz, las clases y cortando el código de un lado al otro. Esperando q al final por un golpe de suerte, todo siga funcionando y los tests sigan pasando, lo cuál es muy poco probable.*

*Lo que se busca al refactorizar es que logremos mejorar el diseño; pero lo más importante, que no hayamos afectado su funcionamiento o introducido errores. Entonces realizar todos estos cambios de un solo golpe y mantener la fe, no nos va ayudar ya que las probabilidades de malograr lo que ya estaba hecho son muy grandes. El clásico miedo a cambiar algo que ya funciona.*

***Tip Número 1 de la refactorización: Realizar Baby – Steps.***

Es decir, realizamos un pequeño cambio rápidamente, cuya probabilidad de malograr algo es muy pequeña y luego probamos, realizamos otro pequeño cambio y volvemos a probar. De esta manera no vamos a interrumpir el trabajo o el funcionamiento de la aplicación simplemente xq nuestra clase no está funcionando.

* Comenzamos a extraer el contenido del método InsertQuarter, primero movemos el contenido del condicional HAS\_QUARTER. (MANUALMENTE). El método se llamará InsertHasQuarter. Probamos que todo sigue funcionando.

Si bien es cierto todo sigue funcionando y hemos realizado baby-steps, este pequeño cambio nos ha involucrado 3 pasos, en los cuales nos hemos podido equivocar. Es por esto que los IDES nos ofrecen alternativas automatizadas para realizar estos refáctorings y que podamos hacerlos de manera más rápida y efectiva.

Por ejemplo utilizando el IDE, seleccionamos la línea de código que queremos extraer, click derecho Refactorings/Extract Method (ALT+SHIFT+T).

TIP 2: Utilizar las refactorizaciones automatizadas, siempre que sea posible.

Particularmente ahora, vamos a tratar de utilizar la mayor cantidad de veces una refactorización automatizada cuando vayamos a hacer algún cambio.

Algunas características de estas herramientas es no es posible utilizarlas para el 100% de los casos por las propias limitantes, pero sí para la gran mayoría de los casos.

Estas herramientas nos suelen ofrecer todos los refáctorings estándar y conocidos, pero estos refáctorings permiten aplicar desde pequeños a medianos cambios. Y no cambios de gran magnitud, como por ejemplo nuestro caso, por lo tanto no vamos a encontrar un Refactoring que sea “Conditionals to Polimorfirm” que nos haga todo el trabajo, ya que se requiere muchas decisiones humanas en este proceso que la herramienta no puede tomar por si misma. Entonces mediante la aplicación de estos pequeños refactorings de manera continua es como logramos conseguir una refactorización de mayor nivel.

El refactoring que acabamos de hacer “Extract Method” es el primero de los refáctorings que vamos a hacer y probablemente sea el más importante de todos, ya que tomando como base este movimiento podremos realizar refactorizaciones más grandes.

## Refactoring Steps

1. Extraer a métodos el InsertQuarter.
2. Aplicar el **“Move”** para mover a una clase cáda método extraido.
   1. Considerar las diferencias entre R# y Eclipse
   2. Considerar que cuando se crear una clase R# la crea en el mismo archivo y tenemos que moverla.
   3. Antes de mover fijarse si el método depende de algún campo interno de la clase. Si depende de alguno, encapsularlo dentro de un get/set para que pueda ser accedido de la otra clase.
3. Crear una interfaz de las clases State ya que todas ellas comparten un mismo método Insert.
   1. Se necesita tener una única firma para todos los métodos pero el método InsertNoQuarter lleva un parámetro. Tenemos 2 opciones:
      1. Crear un constructor donde se le pase el parámetro, agregar los parámetros a través de la refactorización **“Change Signature”**. (Opción elegida)
      2. Que todos los insert reciban como parámetro el GumballMachine.
   2. Eliminar el parámetro del InserNoQuarter.
   3. Ahora que los métodos no tienen ningún parámetro podemos extraer una interfaz. En la primera clase, lo primero que vamos a hacer es un **“Rename”** ya que todos los métodos deben llamarse igual.
   4. Colocamos el método como público y aplicamos el **“Extract Interface”**. (opcional) No seleccionar “Use Interface Where Possible”.
   5. Realizamos lo mismo para todas las clases.
   6. *Para las siguientes extracciones: primero ir método por método para fijarnos si cumple todo y luego crear el método en la interfaz.*
4. Modificar el InsertQuarter para que aproveche la interfaz que hemos creado, podemos observar que todos hacen llamado al state correspondiente, al método Insert. Entonces de alguna manera tenemos código duplicado que podemos eliminar.
   1. Extraemos los estados en un variable CurrentState aplicando el “Extract Variable”.
      1. Eclipse: Observamos que la variable a la cuál aplicamos el ExtractInterface está declarada como State.
   2. Aplicamos un **“Generalize Type”.**
   3. Separamos declaración de asignación.
   4. Subimos la declaración dándole un valor por defecto de null.
   5. Bajamos la llamada al método insert.
   6. *Para las siguientes transformaciones, ya no realizar todos los pasos sino defrente copiar y pegar el contenido de un método anterior y cambiar el nombre de la llamada al método de la clase State.*
5. Extraer los condicionales a un método tipo factoria, de tal manera que ese método sea el encargado de devolverme la clase State correcta de acuerdo al estado en el cuál se encuentra la clase. Aplicamos un **“Extract Method”** y llamamos al método CreateState.
6. Necesitamos remover las constantes de la parte superior de tal manera que solo utilice las clases State que hemos creado. Si nosotros eliminamos directamente estas constantes vamos a producir múltiples errores y como dijimos antes no es una buena idea ya que siempre debemos realizar pasos pequeños. Para esto se pueden utilizar algo que se denominan “estrategias de refactorización” que son sugerencias a seguir para realizar cambios a gran escala.

Una de ellas y la primera que vamos a utilizar es el “Parallel change” que nos dice que no hay que eliminar todo el código antiguo para remplazar el nuevo, sino implementar todo el código nuevo y que conviva simultáneamente con el código viajo y cuando ya este implementado todo el nuevo, recién empezamos a eliminar el código antiguo paulatinamente.

* 1. Declaramos atributos State de manera similar a las constantes. Crear sus métodos de accesos para ese nuevo atributo.
  2. Inicializar la máquina también en el mismo estado.
  3. Modificar el constructor para que haga uso del nuevo estado cuando Count>0.
  4. Dentro de cada clase State, también paralelizamos la asignación del CurrentState.
  5. Empezamos a eliminar el código de manera paulatina.
     1. Modificamos el método CreateState para que haga uso directamente del CurrentState, ya que nosotros hemos paralelizado en todo momento su asignación.
     2. Eliminamos la propiedad int State, nos fijamos primero sus referencias que no está siendo utilizada. Podemos utilizar un **“Inline” Empty.**
     3. Podemos eliminar las constantes, ya que si la propidad ya no se utiliza las constantes tampoco.
  6. Hacemos una limpieza de la clase.

Hemos tomado uno de los casos del libro HeadFirstDesignPatterns, el diseño inicial del libro tenía problemas ya que presentaba problemas de mantenibilidad, hemos refactorizado ese diseño utilizando baby steps y refáctorings automatizados. Asimismo hemos visto la ténica de Parallel Change.

## Refactorings Help

* Refactoring “Move to Class”:
  + Resharper:
    - La nueva clase debe existir.
    - Método estático.
    - La clase debe ser parámetro del método que queremos mover.
    - Método estático a no estático.
  + Eclipse
    - La nueva clase debe existir.