# Extract Local Variable

## Descripción

Consiste en asignar una expresión a una variable local, de esta forma, cualquier referencia a la expresión en el ámbito local se sustituye por la variable.

## Ejemplo

En este caso, vamos a utilizar la clase que se ha definido GameBoardCCJail ya que se puede observar muy fácilmente la necesidad de refactorizar. En la siguiente imagen podemos ver que hay literales que se repiten en la constructora. Si quisiéramos cambiar de literal, tendríamos que hacerlo en cada una de las apariciones, por ello en este caso nos conviene utilizar Extract Local Variable, de esta forma cuando queramos hacer un cambio, simplemente habrá que aplicarlo en la declaración de la variable local.

```
package edu.ncsu.monopoly;


public class GameBoardCCJail extends GameBoard {

    public GameBoardCCJail() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new JailCard(Card.TYPE_CC);
        Card chanceCard1 = new JailCard(Card.TYPE_CHANCE);

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");

        blue1.setPrice(100);
        blue2.setPrice(100);

        blue1.setRent(10);
        blue2.setRent(10);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);

        addCard(ccCard1);

        addCell(cc1);
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
```

Vamos a refactorizar:

```
public class GameBoardCCJail extends GameBoard {

    public GameBoardCCJail() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new JailCard(Card.TYPE_CC);
        Card chanceCard1 = new JailCard(Card.TYPE_CHANCE);

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        final String colorGroup = "blue";
        blue1.setColorGroup(colorGroup);
        blue2.setColorGroup(colorGroup);
```

Una vez hemos refactorizado, podemos ver que nos ha creado una variable local, definida con el literal y ha sustituido todas las apariciones del literal por la variable local. En este caso aparece como si fuera una variable local constante, ya que le hemos pedido que sea así explícitamente, porque en realidad es una constante de la constructora.

# Extract Constant

## Descripción

Consiste en convertir un número o cadena literal en una variable local. De esta forma todos los usos del literal se sustituyen por la constante definida.

## Ejemplo

En este caso, vamos a utilizar la clase que se ha definido para el test UtilityCellTest ya que se puede observar muy fácilmente la necesidad de refactorizar. En la siguiente imagen podemos ver que hay literales que se repiten a lo largo de la clase en los diferentes métodos (p.ej. Utilility 1). Si quisiéramos cambiar de literal, tendríamos que hacerlo en cada una de las apariciones, por ello en este caso nos conviene utilizar Extract Constant, de esta forma cuando queramos hacer un cambio, simplemente habrá que aplicarlo en la declaración de la constante.

```
public class UtilityCellTest extends TestCase {
    GameMaster gameMaster;

    protected void setUp() {
        gameMaster = GameMaster.instance();
        gameMaster.setGameBoard(new GameBoardUtility());
        gameMaster.setNumberOfPlayers(2);
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
    }

    public void testMonopoly() {
        int u1CellIndex = gameMaster.getGameBoard().queryCellIndex("Utility 1");
        gameMaster.movePlayer(0, u1CellIndex);
        gameMaster.getPlayer(0).purchase();
        int u2CellIndex = gameMaster.getGameBoard().queryCellIndex("Utility 2");
        gameMaster.movePlayer(0, u2CellIndex - u1CellIndex);
        gameMaster.getPlayer(0).purchase();
        assertFalse(gameMaster.getPlayer(0).canBuyHouse());
    }

    public void testPlayerAction() {
        UtilityCell cell =
            (UtilityCell) gameMaster.getGameBoard().queryCell("Utility 1");
        int cellIndex = gameMaster.getGameBoard().queryCellIndex("Utility 1");
        gameMaster.movePlayer(0, cellIndex);
        gameMaster.getPlayer(0).purchase();
        gameMaster.switchTurn();
        gameMaster.movePlayer(1, cellIndex);
        cell.playAction();
        int diceRoll = gameMaster.getUtilDiceRoll();
        assertEquals(
```

Vamos a refactorizar:

```
import junit.framework.TestCase;

public class UtilityCellTest extends TestCase {
    private static final String UTILITY_2 = "Utility 2";
    private static final String UTILITY_1 = "Utility 1";
    GameMaster gameMaster;

    protected void setUp() {
        gameMaster = GameMaster.instance();
        gameMaster.setGameBoard(new GameBoardUtility());
        gameMaster.setNumberOfPlayers(2);
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
    }

    public void testMonopoly() {
        int u1CellIndex = gameMaster.getGameBoard().queryCellIndex(UTILITY_1);
        gameMaster.movePlayer(0, u1CellIndex);
        gameMaster.getPlayer(0).purchase();
        int u2CellIndex = gameMaster.getGameBoard().queryCellIndex(UTILITY_2);
        gameMaster.movePlayer(0, u2CellIndex - u1CellIndex);
        gameMaster.getPlayer(0).purchase();
        assertFalse(gameMaster.getPlayer(0).canBuyHouse());
    }

    public void testPlayerAction() {
        UtilityCell cell =
            (UtilityCell) gameMaster.getGameBoard().queryCell(UTILITY_1);
        int cellIndex = gameMaster.getGameBoard().queryCellIndex(UTILITY_1);
        gameMaster.movePlayer(0, cellIndex);
        gameMaster.getPlayer(0).purchase();
        gameMaster.switchTurn();
        gameMaster.movePlayer(1, cellIndex);
```

Una vez hemos refactorizado, podemos ver que nos ha creado un atributo constante de la clase, definida con el literal y ha sustituido todas las apariciones del literal por la variable constante.

# Move method

## Descripción

Consiste en mover el método de clase. Esto surge porque el método está siendo utilizado más por otras clases que la propia clase en la que está definida. Por ello esta refactorización consiste en crear un nuevo método en la clase que más se utiliza con una implementación similar y o bien eliminar el método de la clase anterior o bien simplificarlo para que sea propia de la clase.

## Ejemplo

En la clase UtilityCell, encontramos el método, *getRent*, pero podríamos eliminarlo de ahí y moverlo a la clase Player, ya que el atributo sobre el que trabaja el método anterior (owner), es de dicha clase. Es por eso que se puede mover ese método.

```
public class UtilityCell extends Cell {

	public int getRent(int diceRoll) {
		if(owner.numberOfUtil() == 1) {
			return diceRoll * 4;
		} else if (owner.numberOfUtil() >= 2) {
			return diceRoll * 10;
		}
		return 0;
	}

}
```

Vamos a refactorizar:

```
public class UtilityCell extends Cell {

	public int getRent(int diceRoll) {
		return owner.getRent (diceRoll);
}

public class Owner {
	public int getRent(int diceRoll) {
		if(numberOfUtil() == 1) {
			return diceRoll * 4;
		} else if (numberOfUtil() >= 2) {
			return diceRoll * 10;
		}
		return 0;
	}
}
```

Como podemos observar hemos movido el método de la clase UtilityCell a la clase Owner, manteniendo la interfaz de la clase UtilityCell.

# Inline

## Descripción

Este método de refactorización permite limpiar o simplificar el código, y se utiliza en diversas ocasiones:

- Cuando un método es llamado una sola vez por otro método, y tiene más sentido como un bloque de código
- Cuando una expresión se ve más limpia en una sola línea.

Este método de refactorización consiste en sustituir la referencia a la variable o método con el valor asignado a la variable o la aplicación del método, respectivamente.

## Ejemplo

En este caso, vamos a utilizar la clase que se ha definido GameBooardCCLoseMoney ya que se puede observar muy fácilmente la necesidad de refactorizar. En la siguiente imagen podemos ver que después de realizar la instancia para la variable cc1, sólo se utiliza una vez. Por ello, en este caso nos conviene utilizar el método de refactorización Inline, ya que vamos a poder poner todo en una sola línea.

```
package edu.ncsu.monopoly;


public class GameBoardCCLoseMoney extends GameBoard {
    public GameBoardCCLoseMoney() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MoneyCard("Pay $20", -20, Card.TYPE_CC);
        Card chanceCard1 = new MoneyCard("Pay $30", -30, Card.TYPE_CHANCE);

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");

        blue1.setPrice(100);
        blue2.setPrice(100);

        blue1.setRent(10);
        blue2.setRent(10);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);

        addCard(ccCard1);

        addCell(cc1);
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

Vamos a refactorizar:

```
public class GameBoardCCLoseMoney extends GameBoard {
    public GameBoardCCLoseMoney() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MoneyCard("Pay $20", -20, Card.TYPE_CC);
        Card chanceCard1 = new MoneyCard("Pay $30", -30, Card.TYPE_CHANCE);

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");

        blue1.setPrice(100);
        blue2.setPrice(100);

        blue1.setRent(10);
        blue2.setRent(10);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);

        addCard(ccCard1);

        addCell(new CardCell(Card.TYPE_CC, "Community Chest 1"));
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

Como podemos ver en la imagen anterior, una vez hemos refactorizado, podemos ver que nos la variable ha desaparecido y que se realiza en una sola línea addCell y la instancia al CardCell.

# Convert local variable to field

## Descripción

Este método de refactorización consiste en convertir una variable local en un atributo de la clase. Esto se puede aplicarcuando en los distintos métodos de la clase se define una variable local con la misma función. Tras la refactorización, todos los usos de la variable local se sustituyen por ese atributo.

## Ejemplo

En este caso, vamos a utilizar la clase que se ha definido para el test RailRoadCellTest ya que se puede observar muy fácilmente la necesidad de refactorizar. En la siguiente imagen podemos ver que hay variables locales que se repiten a lo largo de la clase en los diferentes métodos. La duplicidad de código siempre dificultad la legimibilidad y sobre todo el mantenimiento del software, por ello es recomendable en este caso refactorizar con el método Convert local variable to field.

```
package edu.ncsu.monopoly;

import junit.framework.TestCase;

public class RailRoadCellTest extends TestCase {
    GameMaster gameMaster;

    protected void setUp() {
        gameMaster = GameMaster.instance();
        gameMaster.setGameBoard(new GameBoardRailRoad());
        gameMaster.setNumberOfPlayers(2);
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
    }

    public void testPlayerAction() {
        RailRoadCell cell =
            (RailRoadCell) gameMaster.getGameBoard().queryCell("Railroad A");
        int cellIndex = gameMaster.getGameBoard().queryCellIndex("Railroad A");
        gameMaster.movePlayer(0, cellIndex);
        gameMaster.getPlayer(0).purchase();
        gameMaster.switchTurn();
        gameMaster.movePlayer(1, cellIndex);
        cell.playAction();
        assertEquals(
                1500 - cell.getRent(),
                gameMaster.getPlayer(1).getMoney());
        assertEquals(
                1300 + cell.getRent(),
                gameMaster.getPlayer(0).getMoney());
    }

    public void testPurchaseRailroad() {
        assertEquals(0, gameMaster.getPlayer(0).numberOfRR());
        int cellIndex = gameMaster.getGameBoard().queryCellIndex("Railroad A");
        gameMaster.movePlayer(0, cellIndex);
        gameMaster.getPlayer(0).purchase();
        assertEquals(1300, gameMaster.getPlayer(0).getMoney());
        assertEquals(1, gameMaster.getPlayer(0).numberOfRR());
    }

    public void testRent() {
        RailRoadCell rr1 =
            (RailRoadCell) gameMaster.getGameBoard().queryCell("Railroad A");
        int cellIndex1 = gameMaster.getGameBoard().queryCellIndex("Railroad A");
        gameMaster.movePlayer(0, cellIndex1);
        gameMaster.getPlayer(0).purchase();
```

Vamos a refactorizar:

```
package edu.ncsu.monopoly;

import junit.framework.TestCase;

public class RailRoadCellTest extends TestCase {
    GameMaster gameMaster;
    private int cellIndex_A = gameMaster.getGameBoard().queryCellIndex("Railroad A");

    protected void setUp() {
        gameMaster = GameMaster.instance();
        gameMaster.setGameBoard(new GameBoardRailRoad());
        gameMaster.setNumberOfPlayers(2);
        gameMaster.reset();
        gameMaster.setGUI(new MockGUI());
    }

    public void testPlayerAction() {
        RailRoadCell cell =
            (RailRoadCell) gameMaster.getGameBoard().queryCell("Railroad A");
        gameMaster.movePlayer(0, cellIndex_A);
        gameMaster.getPlayer(0).purchase();
        gameMaster.switchTurn();
        gameMaster.movePlayer(1, cellIndex_A);
        cell.playAction();
        assertEquals(
                1500 - cell.getRent(),
                gameMaster.getPlayer(1).getMoney());
        assertEquals(
                1300 + cell.getRent(),
                gameMaster.getPlayer(0).getMoney());
    }

    public void testPurchaseRailroad() {
        assertEquals(0, gameMaster.getPlayer(0).numberOfRR());
        gameMaster.movePlayer(0, cellIndex_A);
        gameMaster.getPlayer(0).purchase();
        assertEquals(1300, gameMaster.getPlayer(0).getMoney());
        assertEquals(1, gameMaster.getPlayer(0).numberOfRR());
    }

    public void testRent() {
        RailRoadCell rr1 =
            (RailRoadCell) gameMaster.getGameBoard().queryCell("Railroad A");

        gameMaster.movePlayer(0, cellIndex_A);
        gameMaster.getPlayer(0).purchase();
        assertEquals(25, rr1.getRent());
```

Una vez hemos refactorizado, podemos observar en la imagen que nos ha creado un atributo de la clase, que en este caso hemos indicado que se inicialice, y hemos sustituido todas sus apariciones.

# Extract superclass

## Descripción

Este método de refactorización consiste en crear una superclase con las características (atributos y métodos) comunes. Esto se puede aplicar cuando dos clases tienen atributos y métodos comunes. De esta forma se evita tener código duplicado y se facilita la mantenibilidad, si hay que cambiar algo será suficiente con cambiarlo solamente en la superclase.

## Ejemplo

En este caso podemos observar que GameBoardCCGainMoney, GameBoardCCJail , GameBoardCCLoseMoney, y GameBoardCCMovePlayer tienen una parte en común en la constructora, que podríamos pensar que la constructora de la superclase podría tener dicha parte en común.

```
public class GameBoardCCGainMoney extends GameBoard {
    public GameBoardCCGainMoney() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MoneyCard("Win $50", 50, Card.TYPE_CC);
        Card chanceCard1 = new MoneyCard("Win $30", 30, Card.TYPE_CHANCE);

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");

        blue1.setPrice(100);
        blue2.setPrice(100);

        blue1.setRent(10);
        blue2.setRent(10);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);

        addCard(ccCard1);
        addCard(chanceCard1);

        addCell(cc1);
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

Sin embargo no se puede porque clases como GameBoardJail tienen la misma superclase y su constructora no tiene nada que ver a las clases antes mencionadas.

```
package edu.ncsu.monopoly;

public class GameBoardJail extends GameBoard {
    public GameBoardJail() {
        super();
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();
        PropertyCell blue3 = new PropertyCell();
        PropertyCell green1 = new PropertyCell();
        PropertyCell green2 = new PropertyCell();
        JailCell jail = new JailCell();
        GoToJailCell goToJail = new GoToJailCell();

        blue1.setName("Blue 1");
        blue2.setName("Blue 2");
        blue3.setName("Blue 3");
        green1.setName("Green 1");
        green2.setName("Green 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");
        blue3.setColorGroup("blue");
        green1.setColorGroup("green");
        green2.setColorGroup("green");

        blue1.setPrice(100);
        blue2.setPrice(100);
        blue3.setPrice(1450);
        green1.setPrice(200);
        green2.setPrice(240);

        blue1.setRent(10);
        blue2.setRent(10);
        blue3.setRent(10);
        green1.setRent(20);
        green2.setRent(20);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);
        blue3.setHousePrice(50);
```

Por ello en este caso se puede crear una superclase para GameBoardCCGainMoney, GameBoardCCJail , GameBoardCCLoseMoney, GameBoardCCMovePlayer donde tengan esa parte en común y que la nueva super clase sea sub clase de GameBoard.

Vamos a refactorizar:

```
package edu.ncsu.monopoly;

public class GameBoardCC extends GameBoard {
        PropertyCell blue1 = new PropertyCell();
        PropertyCell blue2 = new PropertyCell();

    public GameBoardCC() {
        super();


        blue1.setName("Blue 1");
        blue2.setName("Blue 2");

        blue1.setColorGroup("blue");
        blue2.setColorGroup("blue");

        blue1.setPrice(100);
        blue2.setPrice(100);

        blue1.setRent(10);
        blue2.setRent(10);

        blue1.setHousePrice(50);
        blue2.setHousePrice(50);
    }

}
```

```java
package edu.ncsu.monopoly;


public class GameBoardCCGainMoney extends GameBoardCC {
    public GameBoardCCGainMoney() {
        super();

        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MoneyCard("Win $50", 50, Card.TYPE_CC);
        Card chanceCard1 = new MoneyCard("Win $30", 30, Card.TYPE_CHANCE);


        addCard(ccCard1);
        addCard(chanceCard1);

        addCell(cc1);
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

```java
package edu.ncsu.monopoly;


public class GameBoardCCMovePlayer extends GameBoardCC {
    public GameBoardCCMovePlayer() {
        super();

        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MovePlayerCard("Blue 1" , Card.TYPE_CC);
        Card ccCard2 = new MovePlayerCard("Blue 2", Card.TYPE_CC);
        Card chanceCard1 = new MovePlayerCard("Blue 1", Card.TYPE_CHANCE);



        addCard(ccCard1);
        addCard(ccCard2);

        addCell(blue1);
        addCell(cc1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

```java
package edu.ncsu.monopoly;


public class GameBoardCCLoseMoney extends GameBoardCC {
    public GameBoardCCLoseMoney() {
        super();

        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new MoneyCard("Pay $20", -20, Card.TYPE_CC);
        Card chanceCard1 = new MoneyCard("Pay $30", -30, Card.TYPE_CHANCE);


        addCard(ccCard1);

        addCell(new CardCell(Card.TYPE_CC, "Community Chest 1"));
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

```java
package edu.ncsu.monopoly;


public class GameBoardCCJail extends GameBoardCC {

    public GameBoardCCJail() {
        super();

        CardCell cc1 = new CardCell(Card.TYPE_CC, "Community Chest 1");
        JailCell jail = new JailCell();
        CardCell chance1 = new CardCell(Card.TYPE_CHANCE, "Chance 1");

        Card ccCard1 = new JailCard(Card.TYPE_CC);
        Card chanceCard1 = new JailCard(Card.TYPE_CHANCE);



        addCard(ccCard1);

        addCell(cc1);
        addCell(blue1);
        addCell(jail);
        addCell(blue2);
        addCell(chance1);

    }
}
```

Como podemos observar hemos creado una nueva superclase con la parte en común y las subclases tienen en la constructora sólo la parte que les caracteriza.

# Infer generic type argument

## Descripción

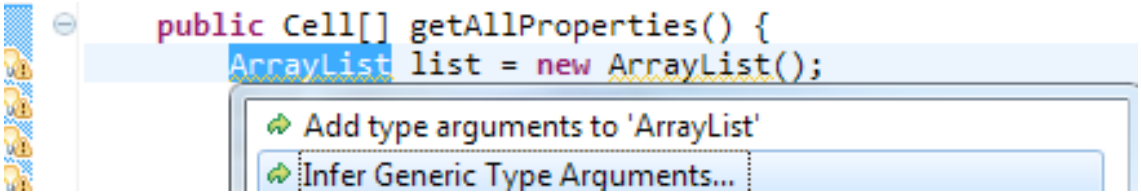
Esta refactorización permite inferir los tipos de los argumentos para las clases genéricas. De esta forma, cuando se esté utilizando una clase genérica, se infiere cuáles son los tipos de los parámetros.

## Ejemplo

El siguiente método es de la clase Player. Como podemos observar nos sale un aviso indicándonos, que se nos ha olvidado definir de que tipo queremos el ArrayList.

```
public Cell[] getAllProperties() {
    ArrayList list = new ArrayList();
    list.addAll(properties);
    list.addAll(utilities);
    list.addAll(railroads);
    return (Cell[])list.toArray(new Cell[list.size()]);
}
```

Vamos a refactorizar:




```
public Cell[] getAllProperties() {
    ArrayList<Cell> list = new ArrayList<Cell>();
    list.addAll(properties);
    list.addAll(utilities);
    list.addAll(railroads);
    return list.toArray(new Cell[list.size()]);
}
```

Como podemos observar el sólo ha detectado que la clase correcta es Cell y han desaparecido los warning.