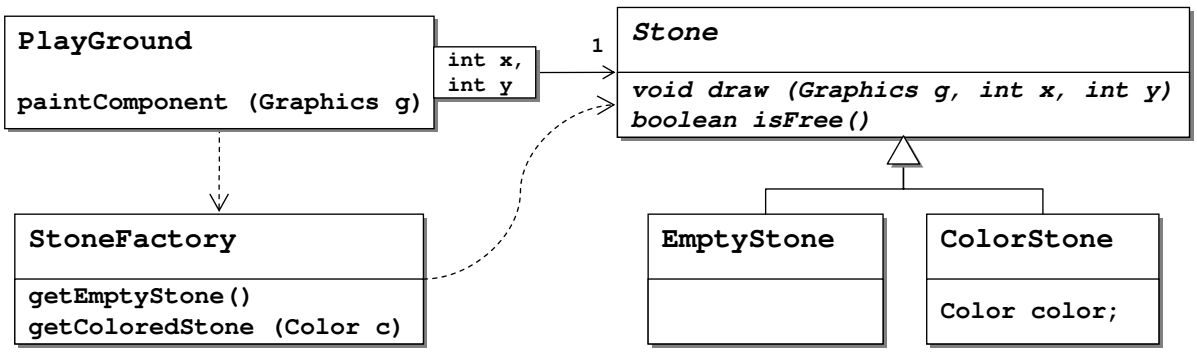
Design Patterns and Frameworks	**Exercise Sheet No. 3**
Dipl.-Inf. Florian Heidenreich	Software Technology Group
INF 2080	Institute for Software and Multimedia Technology
`http://st.inf.tu-dresden.de/teaching/dpf`	Department of Computer Science
	Technische Universität Dresden
	01062 Dresden

Optimization and Control Flow Patterns

Task 1: Tetris Light

A Tetris field is essentially a big collection of stones, some representing empty fields, and some representing parts of a Tetris block, either just falling or landed at the bottom of the play ground some time ago. Whenever a block lands, it dissembles into its individual stones which are considered independent from then on. It makes sense, therefore, to represent each stone by an individual object in an object-oriented Tetris application. Doing so naïvely results in a large amount of objects being allocated, deleted, and moved all the time. What design pattern can be applied to optimize this, taking advantage of the fact that the stones are really very similar?

Solution: We can apply FlyWeight, making each stone a flyweight object, with the only intrinsic state being whether it represents a free field and what color to use when drawing the stone on the screen.



This effectively reduces the number of Stone objects to create to one plus the number of colours to be used.

In the exercise we have discussed a version of Tetris that used only `ColorStone` instances and marked empty cells with a `null` reference.

Task 2: The One and Only Tetris

2a) **Task:**

Imagine a design with a factory for creating the stones in a Tetris application. It makes sense to restrict the number of instances of such a factory that can exist in an application at any one time to at most one. This way we can easily ensure that every part of the application uses the same factory.

Which design pattern can we use for this? What does the corresponding code look like?

Solution: The pattern to be used is the SINGLETON. Its purpose is precisely to restrict the number of instances of a class to at most one.

Here is the relevant code. It uses lazy instantiation, creating the singleton instance only when it is first requested. Note that this code is not multi-threading safe. Two threads calling `getInstance` at almost the same time may receive two different instances of the factory under certain circumstances. There's a pretty good discussion of this issue at `http://www.yoda.arachsys.com/csharp/singleton.html`. Here is another link to a discussion of some more issues with the SINGLETON pattern:
`http://www.javaworld.com/javaworld/jw-04-2003/jw-0425-designpatterns.html`.

```
public class StoneFactory {
  private static StoneFactory s_theFactory;

  public static StoneFactory getInstance () {
    if (s_theFactory == null) {
      s_theFactory = new StoneFactory ();
    }
    return s_theFactory;
  }

  protected StoneFactory () {
    super ();
  }

  ...
}
```

2b) **Task:**

An important variation point for the FACTORY pattern is the specific factory class. We have used this in a prior task to provide for different maze configurations. How can we maintain this variability while also ensuring that no more than one instance of the factory will be used?

Solution: A simple way of setting this statically is by providing the name of the concrete factory class through a configuration file or a system property. The `getInstance()` method could then read this information and use it to instantiate the singleton instance.

If more dynamic approaches are required, an explicit `setClassName()` operation can be provided. To maintain the singleton property—that is, that no more than one instance may exist—the PROXY pattern should be used to hide the actual instance and be able to exchange it when necessary. A problem with this approach is that switching instance classes may invalidate the state of some classes using the current instance. This may be mitigated by sending an event whenever the class is changed.

Task 3: Commanding Tetris

In a Tetris application many different things may happen at any one time: the current stone falls, new stones are generated, user input causes the current stone to move and turn, etc. One way to deal with this multitude of events is by using the COMMAND pattern and providing a central loop which takes commands from a queue and executes them.

3a) **Task:**

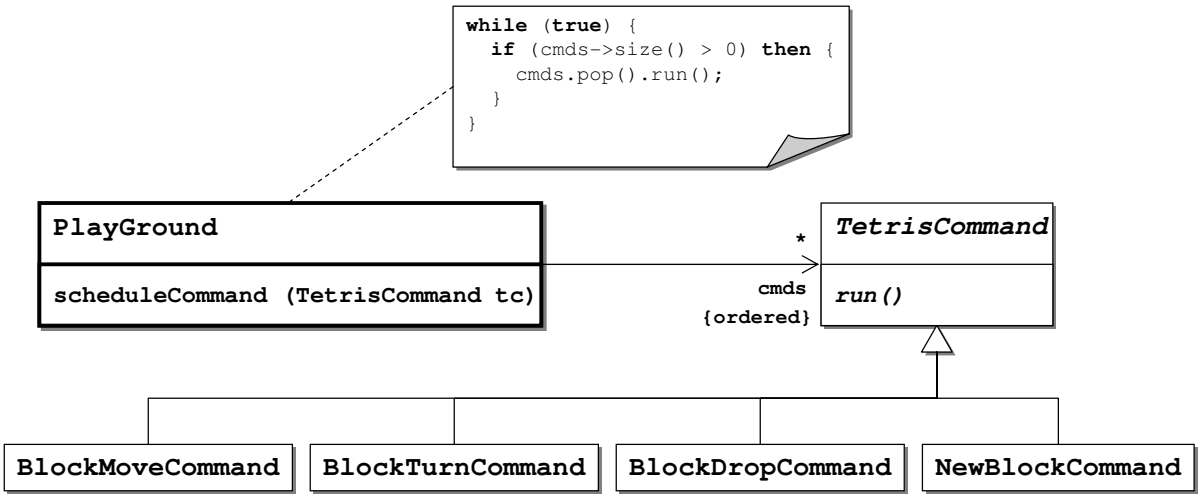
What is the COMMAND pattern and how is it structured?

Solution: *Unfortunately, solution hint is not available.*

3b) **Task:**

Design a Tetris control loop using the COMMAND pattern.

Solution:

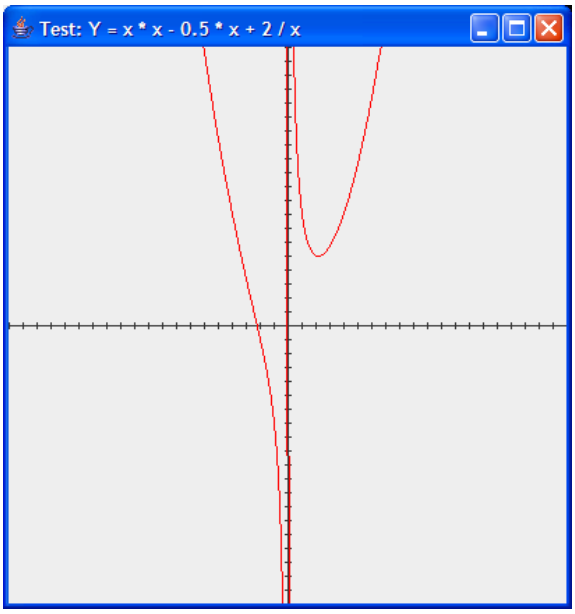


This solution has the advantage of flexibility: It is very easy to add new commands to the game. Because all commands are executed in the same loop, synchronisation happens automatically—no explicit synchronisation is necessary, every command can be implemented without consideration of threading issues.

Also, because commands are reified, we can use them to log the progress of an individual game. It would even be possible to keep a list of all commands so far and use that to undo individual steps in the game.

Task 4: Function Plotter

Design a Swing-based function plotter component that plots the curve corresponding to an expression in a variable 'x' in a range of -10.0 to $+10.0$ (the following figure shows a screen shot of such a component in action).



Use the INTERPRETER pattern to make the function expression a parameter of the new component.

4a) **Task:**

What is the INTERPRETER pattern? What is its structure?

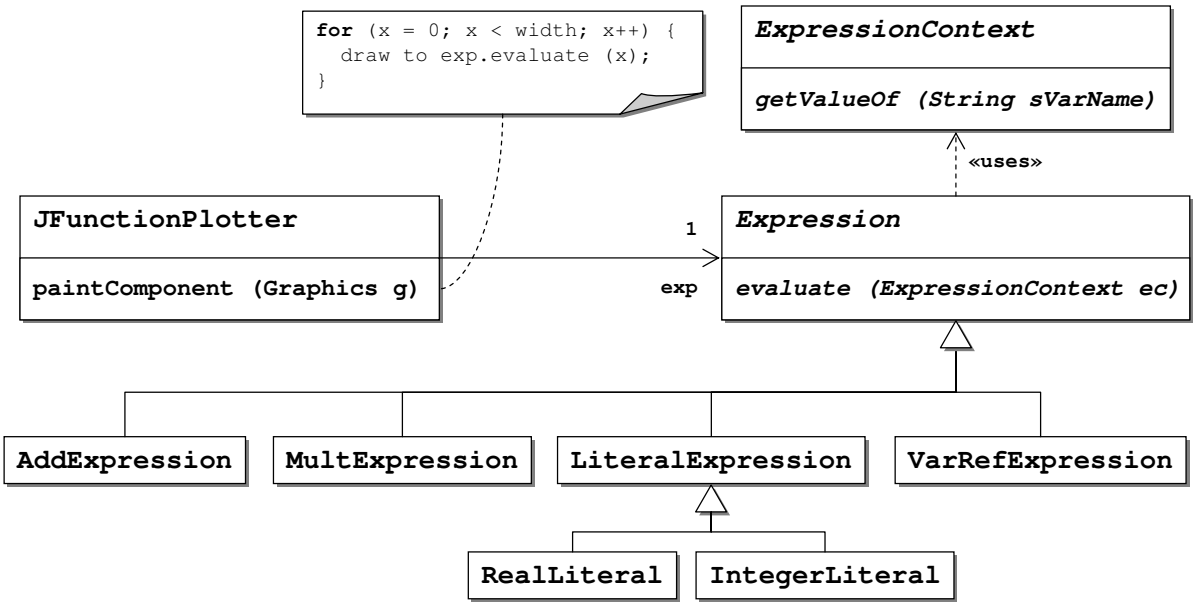
Solution: *Unfortunately, solution hint is not available.*

4b) **Task:**

Develop a design for the new component using the INTERPRETER design pattern. What additional technology do you need to make this system functional?

Hint: To redefine the drawing behaviour of a swing component override `JComponent`'s `paintComponent()` operation and use the operations provided by the `Graphics` parameter so manipulate the output.

Solution: We need to represent expressions in a class hierarchy and provide an `evaluate`-method that determines the current value of an expression given the current value of any variable in the expression.



Then we can use the following code in `paintComponent`:

```
protected void paintComponent (Graphics g) {
  double dXRange = m_dXMax - m_dXMin;
  final double dXScale = dXRange / getWidth ();

  double dYRange = m_dYMax - m_dYMin;
  double dYScale = dYRange / getHeight ();


  // Draw the grid
  int yZero = (int) (getHeight () + m_dYMin / dYScale);
  g.drawLine (0, yZero, getWidth (), yZero);
  for (double dX = 0; dX >= m_dXMin; dX -= m_dXGrid) {
    int xScreen = (int) ( (dX - m_dXMin) / dXScale);
    g.drawLine (xScreen, yZero - 2, xScreen, yZero + 2);
  }
  for (double dX = 0; dX <= m_dXMax; dX += m_dXGrid) {
    int xScreen = (int) ( (dX - m_dXMin) / dXScale);
    g.drawLine (xScreen, yZero - 2, xScreen, yZero + 2);
  }

  int xZero = (int) (-m_dXMin / dXScale);
  g.drawLine (xZero, 0, xZero, getHeight ());
```

```
  for (double dY = 0; dY >= m_dYMin; dY -= m_dYGrid) {
    int yScreen = (int) (getHeight () - ( (dY - m_dYMin) / dYScale));
    g.drawLine (xZero - 2, yScreen, xZero + 2, yScreen);
  }
  for (double dY = 0; dY <= m_dYMax; dY += m_dYGrid) {
    int yScreen = (int) (getHeight () - ( (dY - m_dYMin) / dYScale));
    g.drawLine (xZero - 2, yScreen, xZero + 2, yScreen);
  }

  // Plot the function
  g.setColor (Color.RED);

  final int[] x = {0 };
  Expression.ExpressionContext ec = new Expression.ExpressionContext () {
    public double getVariableValue (String sName) {
      if (sName.equals ("x")) {
        return x[0] * dXScale + m_dXMin;
      } else
        throw new IllegalArgumentException ("Variable " + sName
            + " not defined.");
    }
  };

  int lastY = 0;

  for (; x[0] <= getWidth (); x[0]++ ) {
    double curValue = m_expFunction.evaluate (ec);

    int curY = (int) (getHeight () - (curValue - m_dYMin) / dYScale);

    if (x[0] != 0) {
      g.drawLine (x[0] - 1, lastY, x[0], curY);
    }

    lastY = curY;
  }
}
```

where `m_expFunction` is the member variable holding the expression to be plotted.

In addition, we will need to implement a parser that takes a string-represenation of an expression and turns it into an `Expression` instance. For this, we can use standard parser generators—for example, SableCC or JACC.