Design Patterns and Frameworks	**Exercise Sheet No. 5**
Dipl.-Medieninf. Christian Piechnick	Software Technology Group
INF 2080	Institute for SMT
christian.piechnick@tu-dresden.de	Department of Computer Science
	Technische Universität Dresden
	01062 Dresden

Extensibility Patterns: Extension Access

Task 1: Lively Iteration

Cellular automata [1] are mathematical 'machines' consisting of a grid of individual so-called 'cells', each of which can be in one of k states (typically called 'colours'). A cellular automaton evolves in so-called generations: The state of all cells is updated simultaneously following rules based on the colours of the cells in a certain neighbourhood of the currently evolving cell. Different types of cellular automata exist; they vary in the shape and size of the grid, the number of possible colours per cell, the size and shape of a cell's neighbourhood, and the evolution rules.

1a) **Task:**

Design and implement a class `CellGrid` that represents a grid of cells of a cellular automaton. Provide an interface for access to the individual cells, but hide the structure of the grid (number of dimensions, structure and size of neighbourhood, etc.) from clients.

What design pattern can you use? How do you use it?

Solution: We use ITERATOR to allow clients to access each cell in the grid without having to disclose anything about the structure of the grid.

Everything the client needs to know about an individual cell is made available through the iterator's interface:

```
public interface CellIterator {

  /**
   * Move the iterator to the next cell, if any.
   *
   * @return true, if there still is a cell to be considered, false if all
   *         cells have been treated.
   */
  public boolean next();

  /**
   * @return the colour of the current cell encoded as a non-negative number
   */
  public int getColour();

  /**
   * @return the number of neighbours of a certain colour of the current cell
   */
  public int getNeighbours(int withColour);

  /**
   * Determine the colour of the current cell in the next generation of the
   * automaton
   */
  public void setColourInNextGeneration(int futureColour);

}
```

`CellGrid` then provides access to individual cells only through an instance of `CellIterator`:

```
/**
 * A grid of cells in a cellular automaton. This grid can be of any size and
 * shape. A cell grid stores two generations of a grid, namely the current
 * generation and an image of the next generation. {@see #nextGeneration()} can
 * be invoked to make the future generation the current one.
 */
public interface CellGrid {
  /**
   * @return an iterator of all cells in the current generation of this grid.
   */
  public CellIterator allCells();

  /**
   * Switch to the next generation.
   */
  public void nextGeneration();

}
```

1b) **Task:**

Using the `CellGrid` class from above, design and implement a class `GridChanger` that realises Conway's Game of Life [2]. Life is played on a two-dimensional grid, with cells that are either black ("dead") or white ("alive"). The neighbourhood of a cell consists of the 8 cells directly adjacent to it. For each cell, the colour of the cell in the next generation depends on the cell's current colour and on the number of living (i.e., white) cells in its neighbourhood:

- The cell *dies* (i.e., turns black in the next generation) if it is alive and there are less than 2 or more than 3 living cells in the neighbourhood.
- The cell *survives* (i.e., stays white in the next generation) if it is currently alive and there are 2 or 3 living cells in the neighbourhood.
- The cell *is born* (i.e., turns white in the next generation) if it is currently dead and there are exactly 3 living cells in its neighbourhood.
- The cell *stays dead* (i.e., stays black in the next generation) if it is dead and there are more or less than 3 living cells in its neighbourhood.

Solution:

```
public class GridChanger implements Runnable {

  private CellGrid cellGrid;

  public GridChanger(CellGrid cellGrid) {
    super();
    this.cellGrid = cellGrid;
  }

  @Override
  public void run() {
    while (true) {
      try {
        Thread.sleep(500);
      } catch (InterruptedException e) {
      }

      step();
    }

  }

  public void step() {
    for (CellIterator ci = cellGrid.allCells(); ci.next();) {
      if (ci.getColour() == 1) {
        switch (ci.getNeighbours(1)) {
        case 0:
        case 1:
```

```
          case 4:
          case 5:
          case 6:
          case 7:
          case 8:
            ci.setColourInNextGeneration(0);
            break;

          default:
            ci.setColourInNextGeneration(1);
          }
        } else {
          ci.setColourInNextGeneration((ci.getNeighbours(1) == 3) ? 1 : 0);
        }
      }

      cellGrid.nextGeneration();
    }
}
```

1c) **Task:** *

Use additional patterns (STRATEGY, OBSERVER, INTERPRETER, ...) to implement a completely generic cellular automaton grid. Parametrize size and structure of the grid, size and structure of a cell's neighbourhood, number of colours, and the rules for determining a cell's colour in the next generation.

This task is of added complexity. We may not discuss it in the exercise, but I will be happy to comment on any solution of yours that you send me.

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

Bibliography

1. Eric W. Weisstein. *Cellular Automaton.* From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/CellularAutomaton.html`
2. Eric W. Weisstein. *Life.* From MathWorld—A Wolfram Web Resource. `http://mathworld.wolfram.com/GameofLife.html`

Task 2: Extensible Insurance Contracts

Insurance contracts are very long-lived documents that are treated by many different people for many different reasons during their life-time. In a software system for the management of insurance contracts, each of these clients needs a custom interface to the insurance-contract object. Sometimes the same person needs to treat very different types of insurance contracts in the same manner. Occasionally, new types of treatment need to be added dynamically, without affecting any of the pre-existing code.

2a) **Task:**

Understand the design pattern Extension Object [1]. What are its elements? How do they collaborate to support solving the above problems?

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

2b) **Task:**

Use the Extension Object pattern to design and implement an insurance contract management system. Support the following roles:

- Initialization: The contract object has just been created and needs to be filled with the correct data.

- Conclusion: The contract has been accepted and needs to be signed by all parties.
- Termination: The contract's duration has passed, all the money goes to the company :-)

Solution: The following code provides the basic structure, implementing only the initialization role, however.

```
/*
 * InsuranceContract.java
 *
 * History
 * ———————
 *
 * 28.10.2005 Zschaler Created.
 */
package de.tudresden.inf.wwwst.extensionobject;

import java.util.Map;
import java.util.TreeMap;

/**
 * An insurance contract.
 *
 * @author Zschaler
 * @since 28.10.2005
 */
public class InsuranceContract {

  /**
   * Final value of the contract.
   */
  private double m_dFinalValue;

  /**
   * Monthly payments.
   */
  private double m_dMonthlyDue;

  /**
   * Duration of the contract in years.
   */
  private long m_lDuration;

  /**
   * Name of the person for whom the contract has been issued.
   */
  private String m_sOwner;

  /**
   * Create a new empty contract.
   */
  public InsuranceContract () {
    super ();

    class InitRole extends ICRoleAdapter implements ICInitRole {
      public void setParameters (String sOwner, double dFinalValue,
          long lDuration) {
        m_sOwner = sOwner;
        m_dFinalValue = dFinalValue;
        m_lDuration = lDuration;
        // Calculate monthly payments and don't forget the revenue!
        m_dMonthlyDue = (dFinalValue + 10000.00) / (lDuration * 12);
      }

      public double getMonthlyDue () {
        return m_dMonthlyDue;
      }
    };

    // Set up the roles of this contract
    addRole ("INITIALIZE", new InitRole ());
```

```
}

/**
 * Tag interface common to all roles an insurance contract may play.
 *
 * @author Zschaler
 * @since 28.10.2005
 */
public interface ICRole {
  /**
   * @return the contract whose role this is.
   */
  public InsuranceContract getContract ();
}

/**
 * A role for initializing a contract.
 *
 * @author Zschaler
 * @since 28.10.2005
 */
public interface ICInitRole extends ICRole {
  /**
   * Set the contract's parameters.
   */
  public void setParameters (String sOwner, double dFinalValue, long
  lDuration);

  /**
   * Get the monthly payment of this contract as computed by
   * {@link #setParameters(String, double, long)}.
   */
  public double getMonthlyDue ();
}

/**
 * Helper class.
 * @author Zschaler
 * @since 28.10.2005
 */
public abstract class ICRoleAdapter implements ICRole {
  public InsuranceContract getContract () {
    return InsuranceContract.this;
  }
}

/**
 * Exception thrown when a requested role is not available.
 *
 * @author Zschaler
 * @since 28.10.2005
 */
public static class UnsupportedRoleException extends Exception {
  public UnsupportedRoleException (String sRoleName) {
    super ("Unsupported role: <" + sRoleName + ">");
  }
}

/**
 * The set of roles supported by this contract.
 */
private Map<String, ICRole> m_mpRoles = new TreeMap<String, ICRole> ();

/**
 * Get a role with a certain name.
 */
public ICRole getInterface (String role) throws UnsupportedRoleException {
  ICRole result = m_mpRoles.get (role);

  if (result == null) {
```

```
      throw new UnsupportedRoleException (role);
    }

    return result;
  }

  /**
   * Register a role with a certain name.
   */
  protected void addRole (String sRoleName, ICRole role) {
    m_mpRoles.put (sRoleName, role);
  }
}
```

2c) **Task:**

What do you have to do to support another role "Incident" (i.e., the incident against which the insurance is held, has occurred)?

Solution: All that's required is a new implementation of `ICRole` with the corresponding functionality. As soon as this has been registered with the insurance-contract object, the new role can be used.

2d) **Task:**

How can different document types (for example, insurance contracts, but also letters, etc.) support the same role?

Solution: By providing an interface `IExtensible` the only purpose of which it is to provide access to extensions through a `getRole` object. If roles are given globally unique identifiers, any object that implements this interface can then be used in the same way, independent of its class. This even permits virtual dynamic class changes.

An implementation of this is in Microsoft's most basic COM interface `IUnknown`. This interface provides two features: reference counting and access to other interfaces of a COM object using globally unique interface identifiers.

Bibliography

1. Gamma, E. 1997. *Extension object.* In Pattern Languages of Program Design 3, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley Software Pattern Series. Addison-Wesley Longman Publishing Co., Boston, MA, 79–88.
   Also at http://st.inf.tu-dresden.de/Lehre/WS06-07/dpf/gamma96.pdf
   A nice tutorial also exists at http://www.design-nation.net/en/archives/000488.php

Task 3: Discussion of Patterns

3a) **Task:** Enumerate 4 different types of PROXIES, and tell what they do.

Solution:

1. A *remote proxy* provides a local representative for an object in a different address space.
2. A *virtual proxy* creates expensive objects on demand.
3. A *protection proxy* controls access to the original object. Protection proxies are useful when object should have different access rights.
4. A *smart reference* is a replacement for a bare pointer that performs additional actions when an object is accessed. Typical uses include:
   - counting the number of references to the real object for *garbage collection*
   - checking for *locking*

This list has been taken directly from the GOF book.

3b) **Task:** Compare ADAPTER and BRIDGE. Enumerate commonalities and differences.

Solution: ADAPTER and BRIDGE have a few things in common:

- Both enhance flexibility by introducing an additional indirection into the access to an object.
- Both hand on commands from an interface that the object originally did not implement.

The key difference between the two patterns can be found in their intents. ADAPTER focuses on resolving incompatibilities between two existing interfaces. It does not care for the implementation of these interfaces, nor does it care about the possible development of class hierarchies. ADAPTER couples two classes developed independently. Their cooperation has originally not been planed or predicted.

Users of BRIDGE, however, know from the beginning that there will be multiple implementations for the same abstraction.

3c) **Task:** Compare BRIDGE and TEMPLATEMETHOD. Enumerate commonalities and differences.

Solution: TEMPLATE METHOD and BRIDGE both declare methods in abstract superclasses and implement them in subclasses.

The key difference is that in the abstract class of TEMPLATE METHOD, one method is implemented. This breaks the complete separation of abstraction and implementation intended by BRIDGE. Abstraction and implementation can no longer be refined independently.

TEMPLATE METHOD uses operations from the same object only. BRIDGE uses operations from another object. This can lead to runtime errors in weakly typed languages (e.g., SmallTalk).

3d) **Task:** Enumerate the cases in which VISITOR can be employed. Characterize the advantages of the pattern.

Solution: VISITOR can be used to advantage, when an object structure with many classes must be traversed, and class specific operations are to be executed on each object.

VISITOR helps group into one class operations that belong together semantically, even though they work on different classes. This helps avoid cluttering data classes with lots of operations for different use cases. All operations of the same use case are encapsulated in one VISITOR.

Additionally, VISITOR is perfect when the data structure changes rarely, but new operations are added frequently.

VISITOR is often used together with COMPOSITE.

3e) **Task:** Compare TemplateMethod and Strategy. What are commonalities, what are differences?

Solution: In Template Method, the algorithm is fix, but some parts are variable. On the other hand, in Strategy, the algorithm is variable, and the client is fix.