Investigating a New Design Pattern for Efficient Implementation of Prediction Algorithms

Arpad Gellert, Adrian Florea
Department of Computer Science and Electrical Engineering
Lucian Blaga University of Sibiu
Emil Cioran Street, No. 4, 550025 Sibiu
Romania
{arpad.gellert, adrian.florea}@ulbsibiu.ro

Journal of Digital Information Management

ABSTRACT: *In our previously published researches we used different prediction algorithms to solve several problems in computer architecture and ubiquitous computing. During the time, we observed what is common among different solutions and also what can differentiate these solutions. Therefore, due to our experience in designing predictors, we are able now to propose a simple and efficient general solution for any problem that implies prediction. Since prediction is a widely used technique in many fields, our proposed design pattern can be very useful for software developers.*

Categories and Subject Descriptors:
K.6.3 [Software Management]; **I.3.5 [Computational Geometry and Object Modeling]**; **G.1 [Numerical Analysis]**; Numerical Algorithms

General Terms:
Software Design; Object Modelling, Design Pattern, Prediction Algorithms

Keywords: Design Patterns, Predictor, Reusable Object-oriented Software, Markov Chain, Multi-Layer Perceptron

Received: 12 June 2013, Revised 19 July 2013, Accepted 27 July 2013

1. Introduction

The general prediction mechanism consists in anticipating future contexts based on current and previous context information, recovering the correct context if the speculation fails and updating the predictor to improve future prediction accuracy. Prediction can be very useful if the availability of some data in advance allows to reduce waiting times, improving thus the efficiency. Obviously, the prediction must be accurate, because in some applications a misprediction has costs due to the necessity of correct state recovery. The quality of a prediction model is highly dependent on the quality of the available data. Especially the choice of the features to base the prediction on is important. In our previous works, we focused on prediction algorithms applied to solve several problems in computer architecture (branch prediction, register value prediction, load value prediction) and ubiquitous computing (person movement prediction). During the time, we have designed and used different high complexity prediction methods based on Markov chains, Hidden Markov Models (HMM), Neural Networks, and also some simple methods which are very efficient for hardware implementation like the Last Value Predictor and the Two Level Predictors. Prediction is a widely used technique in many fields and therefore we propose a useful implementation solution. Thus, the aim of this paper is to introduce the Predictor design pattern, to describe it and to illustrate an example of implementation. This design pattern is not only focused on micro architecture and ubiquitous computing, on the contrary, is a solution for any subdomain of Computer Science which is using pattern recognition, classification methods, etc. As far as we know, we are the first researchers who define a design pattern for predictors.

The organization of the rest of this paper is as follows. In Section 2 we review the related work in the fields of prediction and design patterns. Section 3 describes our proposed design pattern. In Section 4 we present and explain code fragments of the Predictor implementation

in Java and we also illustrate the experimental results. Section 5 concludes the paper.

2. Related Work

Several works [4] [19] [2] [10] present simple and elegant solutions to specific problems in object-oriented software design. Design patterns are solutions that have been developed over time. These solutions are maximizing both the reuse and the flexibility in software. The idea of a Predictor design pattern is based on the fact that prediction mechanisms are used in many applications, some of them previously designed by us.

In previous works [15] [5] we designed neural-, Markov- and HMM-based predictors to anticipate the next movements of persons. The application predicts the next room based on the history of rooms, visited by a certain person moving within an office building. These predictors were evaluated by some movement sequences of real persons, acquired from the Smart Doorplates project developed at University of Augsburg [13]. The simulation results have shown accuracy in next location prediction reaching up to 92%. Other neural network approaches used in ubiquitous systems were presented by [1] and [12].

In other research papers [17] [6] we used different prediction methods in order to anticipate the behavior of branch instructions which appear in high level program constructs like if, switch, for, while, etc., and are a major bottleneck in the instruction-level parallelism (ILP) exploitation of multiple instruction issue microprocessors. During the time, several prediction methods have been developed based on some well-known learning algorithms (Markovian, neural, Bayesian, decision trees, support vector machine, etc.) simplified for efficient hardware implementation. Through dynamic branch prediction, microprocessors are speculatively processing multiple basic blocks in parallel and therefore their ability to increase ILP is stronger. For the evaluations, the predictors were implemented in software simulators and tested on the SPEC 2000 benchmarks. We used [18], among other metrics, a HMM-based prediction algorithm to evaluate the random degrees of some difficult to predict branches and we have shown that these branches have intrinsic random behavior, being generated by very complex program structures.

In other works [16] [7] [8] [9] we proposed and implemented several value prediction methods in order to increase ILP in superscalar and simultaneous multithreading microarchitectures. The idea of all these methods was to unlock some dependent instructions by anticipating either the register values or the results of the long latency load instructions. All these predictors were implemented in software simulators in order to evaluate their prediction accuracy, but also the overall processing performance and energy consumption, two very useful metrics from computer architect viewpoint. Some of these predictors proved to be efficient regarding both objectives [8].

All the predictors mentioned in this section fit the same general solution, being differentiated just by the way the prediction is performed, therefore we consider that presenting a design pattern for them could be very useful.

3. Describing the Predictor Design Pattern

We provide in this section a description of the Predictor design pattern.

3.1 Intent

Provides an implementation template for any prediction mechanism. It can be used to anticipate future states or symbols in a software application and it is useful when speculatively knowing future states or symbols in advance unlocks some application-specific dependencies involving thus an execution speedup.

3.2 Motivation

There are many situations when the behavior of a software process in a certain context is always or mostly the same. In such cases, if we record the history of contexts and the associated behaviors, when we identify those contexts in the future, we can predict with high accuracy the corresponding behaviors. Obviously, it is possible to encounter mispredictions, but a confidence mechanism can help to decide when to predict or when to avoid prediction, increasing thus the overall accuracy.

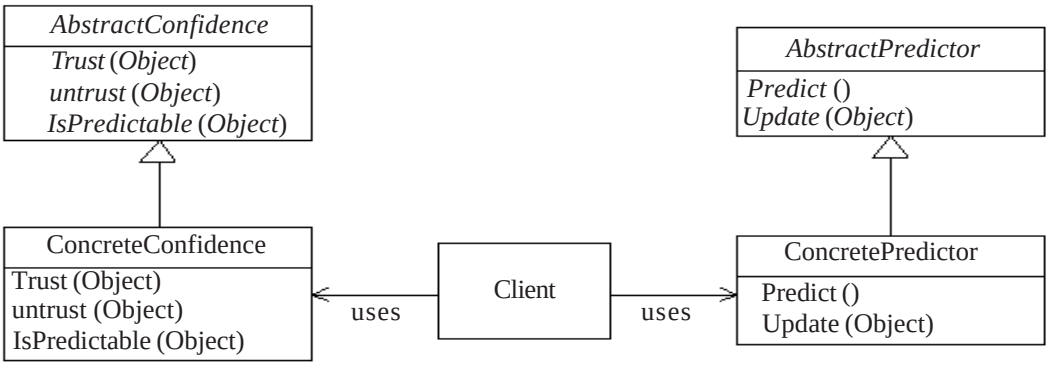

Figure 1. The structure of the Predictor design pattern

3.3 Applicability
The Predictor design pattern can be used in applications which have prediction processes. It helps to efficiently implement prediction mechanisms.

3.4 Structure
Figure 1 presents the class diagram of a system which is using the Predictor design pattern.

3.5 Participants
- *AbstractConfidence*
- *ConcreteConfidence*
- *AbstractPredictor*
- *ConcretePredictor*
- *Client*

3.6 Collaborations
AbstractConfidence relies on its subclass to define the *Trust*, *Untrust* and *IsPredictable* methods. *AbstractPredictor* relies on *ConcretePredictor* to define the *Predict* and *Update* methods. The prediction is performed as follows. The client checks if the current context is predictable by calling the *IsPredictable* method of *ConcreteConfidence*. Then the client calls the *Predict* method of *ConcretePredictor* and the returned prediction is used in advance to unlock dependencies only if the current context is predictable. When the real symbol is known, the client compares it with the predicted one and correspondingly performs the update by calling the *Update* method of *ConcretePredictor*, which introduces the new symbol to the recorded history of symbols, and also by calling the *Trust* method of *ConcreteConfidence* in case of correct prediction or the *Untrust* method in the misprediction case.

3.7 Consequences
The Predictor design pattern eliminates stalls by anticipating future states based on current and previous context information. A potential disadvantage of using a predictor can occur in some applications if the prediction accuracy is low due to the high number of mispredictions and the recovery is time consuming.

3.8 Implementation
In this implementation the update of the predictor is performed by the client and it consists in calling the *Trust* or *Untrust* method of the *Confidence* and the *Update* method of the *ConcretePredictor*. If the number of possible observations is very high we recommend the use of hash tables to keep prediction information.

3.9 Known uses
Markov- and HMM-based predictors [5] [17-18] and Neural Networks [15].

AbstractPredictor is an interface which relies on a predictor class to define the *predict* and *update* methods. The *predict* method anticipates the next observation based on the history of observations. The *update* method must actualize the predictor by maximizing the probability of correct predictions for the future. The *AbstractPredictor* is defined as follows:

4. An Example of Predictor Implementation

To predict or anticipate a future situation, learning techniques as Markov Chains, Hidden Markov Models, Bayesian Networks, Time Series or Neural Networks are obvious candidates. The challenge is to adapt such algorithms to work with context information. In this section we present a Markov chain used by [5] and also a Multi-Layer Perceptron (MLP) used by [15], both implemented in Java to predict the movements of employees within an office building. The goal of the research was to design some smart doorplates that are able to direct visitors to the current location of an office owner based on a location-tracking system and predict if the office owner is soon coming back.

4.1 Person Movement Prediction
The application just generates statistics regarding person movement prediction and reports the number of predictions, the number of correct predictions and also the prediction accuracy. Figure 2 presents the class diagram of the application.

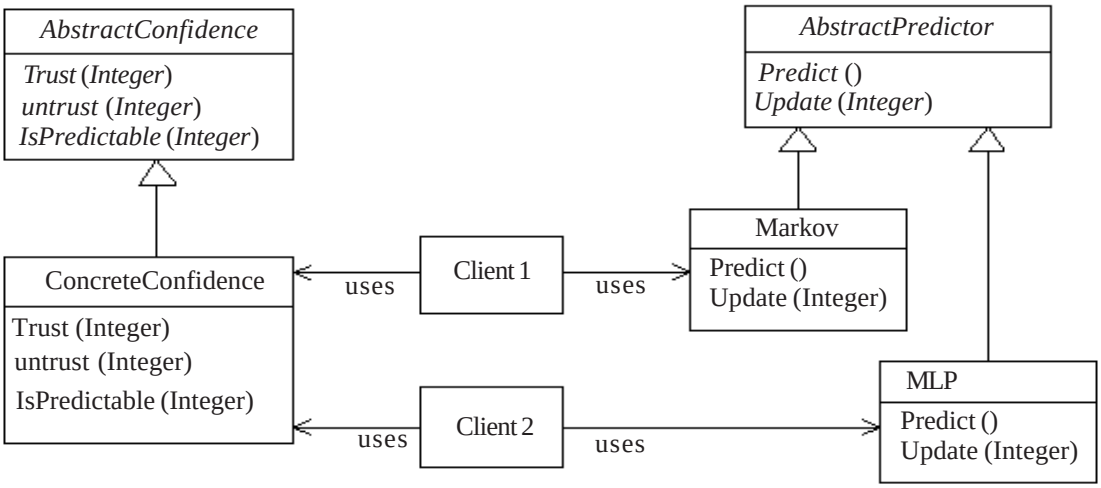

Figure 2. The structure of the person movement predictor

```
public interface AbstractPredictor {
  public abstract Integer predict ();
  public abstract void update (Integer observation);
}
```

The *Markov* class is a concrete Markov-chain-based predictor which defines the *predict* and *update* methods of the *AbstractPredictor* interface. The *predict* method identifies the context consisting in the last *R* (input parameter) observation symbols, searches for it in the observation sequence, determines which observation symbol followed the context with the highest frequency and returns that symbol as the predicted next observation. A transition-table-based implementation is also possible, but it is inefficient for a high number of observation symbols. The *update* method adds the real observation symbol, when it is available, to the observation sequence. More details about Markov predictors were presented by [14]. The *Markov* class is presented below:

The *MLP* class is another concrete predictor, a Multi-Layer Perceptron with one hidden layer and is using the backpropagation learning algorithm. In general, the number of neurons in the input and output layers depends on the

```
public class Markov implements AbstractPredictor{
  int P [];              //the observation symbol probability distribution
  java.util.ArrayList Q = null; //the observation sequence
  int R = 0;                    //the order of the Markov Chain
  int N = 0;                    //number of observation symbols
  int T = 0;                    //length of observation sequence
  int context [];               //prediction context

  public Markov (int nObservationSymbols, int order) {
    R = order;                          // the order of the Markov Chain
    context = new int [R];
    N = nObservationSymbols;        // the number of distinct observation symbols
    Q = new java.util.ArrayList ();
  }

  public Integer predict (){
    P = new int [N];
    T = Q.size ();
    For (int k = 0; k < R; k++)
      Context [k] = ((Integer) Q.get (T – R + k)).intValue ();
    for (int i = R; i < T; i++){
      boolean is Context = true;
      for (int k = 0; k < R; k++)
        if (((Integer) Q.get(i – R+k)).intValue () != context [k]){
          isContext = false;
          break;
        }
      if (isContext)
        P [((Integer)Q.get(i)).intValue ()]++;
    }
    int pred = 0;
    int max = P [0];
    for (int k = 1; k < N; k++)
      if (P [k] > max){
        max = P [k];
        pred = k;
      }
    return new Integer (pred);
  }

  public void update (Integer observation){
    Q.add (observation);
  }
}
```

representation of the problem. In this application we chose binary encoding for the input layer and one-room-one-neuron encoding for the output layer. The *MLP* class defines the *predict* and *update* methods of the *AbstractPredictor* interface. Beside these methods, it contains some MLP-specific methods like *generateRandomWeights*, *F*, *dF*, *forward* and *backward*, but also some methods to codify the observation symbols to fit the input layer or the output layer, *decimalToBinary* and

decimalToCode, respectively. For binary coded inputs and outputs, the uni-polar sigmoid activation function can be used:

$$F(x) = \frac{1}{1 + e^{-x}} \qquad (1)$$

Since in the presented application the inputs and outputs are codified with -1 and 1, the following bi-polar sigmoid activation function was considered:

$$F(x) = \frac{1 - e^{-x}}{1 + e^{-x}} \qquad (2)$$

The activation function is defined in the *F* method and its derivate in *dF*. The *generateRandomWeights* method is used to randomly initialize the weights in the $[-2/N, 2/N]$ interval, where N is the number of input layer neurons [3]. More details about the backpropagation algorithm used in MLPs are given by [11]. The definition of the *MLP* class is presented below:

```
import java.lang.Math;
public class MLP implements AbstractPredictor{
 java.util.ArrayList Q = null;          //the observation sequence
 private int history;
 private int nNeuronsForSymbol;
 int input [];                                  //array of input values
 private int nInputLayerNeurons;
 private int nHiddenLayerNeurons;
 private int nOutputLayerNeurons;
 private double neth [];              //the hidden layer values before activation
 private double whin [] [];           //hidden-input weight matrix
 private double bhin [];              //hidden layer bias array
 private double hidd [];              //hidden layer values after activation
 private double neto [];              //the output values before activation
 private double wohi [] [];           //output-hidden weight matrix
 private double bohi [];              //output layer bias array
 public double out [];                          //the output values after activation
 private double deltaout [];                    //the output layer error terms
 private double deltain [];           //the hidden layer error terms
 private double learningRate = 0.3;

  public MLP (int nNeuronsForSymbol, int m, int p, double learningRate, int history){
  Q = new java.util.ArrayList ();
  this.nNeuronsForSymbol = nNeuronsForSymbol;
   this.history = history;
  input = new int[history*nNeuronsForSymbol];
  this.nInputLayerNeurons = nNeuronsForSymbol* history;
  this.nHiddenLayerNeurons = m;
   this.nOutputLayerNeurons = p;
  this.learningRate = learningRate;
   neth = new double [nHiddenLayerNeurons];
   whin = new double [nHiddenLayerNeurons][nInputLayerNeurons];
   bhin = new double [nHiddenLayerNeurons];
   hidd = new double [nHiddenLayerNeurons];
   neto = new double [nOutputLayerNeurons];
   wohi = new double [nOutputLayerNeurons] [nHiddenLayerNeurons];
   bohi = new double [nOutputLayerNeurons];
   out = new double [nOutputLayerNeurons];
   deltaout = new double [nOutputLayerNeurons];
deltain = new double [nHiddenLayerNeurons];
   generateRandomWeights ();
  }
```

```
 private void generateRandomWeights (){
 double wi = 4.0/nInputLayerNeurons;        //weight interval
 double hwi = 2.0/nInputLayerNeurons;       //half weight interval
 for (int j = 0; j < nHiddenLayerNeurons; j++) {
  bhin [j] = ((Math.random ()*10000)% (Math.floor (wi*100)))/100.0 – hwi;
   for (int k = 0; k < nInputLayerNeurons; k++)
    whin [j] [k] = ((Math.random()*10000)% (Math.floor(wi*100)))/100.0 – hwi;
 }
 for (int j = 0; j < nOutputLayerNeurons; j++){
  bohi [ j] = ((Math.random ()*10000)% (Math.floor (wi*100)))/100.0 – hwi;
   for (int k = 0; k < nHiddenLayerNeurons; k++)
    wohi [j] [k] = ((Math.random ()*10000)% (Math.floor (wi*100)))/100.0 – hwi;
 }
}

private double F (double x){
 return (1 - Math.exp(-1 * x))/(1 + Math.exp (–1 * x));
}

private double dF (double x){
 return (1 – (F(x)*F(x)))/2;
}

public void forward (int in []){
  /* hidd <- in */
  int j, l;
 for (j = 0; j < nHiddenLayerNeurons; j++){
   neth [j] = bhin [j];
   for (l = 0; l < nInputLayerNeurons; l++)
    neth [j] + =  whin [j] [l] * in [l];
   hidd [j] = F (neth [j]);
 }
  /* out <- hidd */
 for (j = 0; j < nOutputLayerNeurons; j++){
   neto [j] = bohi [j];
   for (l = 0; l < nHiddenLayerNeurons; l++)
    neto [j] += wohi [j] [l] * hidd [l];
   out [j] = F (neto [j]);
 }
}

public void backward (int tp [], int in []){
  /* out -> hidd */
 for (int j = 0; j < nOutputLayerNeurons; j++)
   for (int l = 0; l < nHiddenLayerNeurons; l++){
    deltaout [ j] = (tp [ j] – out [ j]) * dF (neto [ j]);
    wohi [ j] [l] + = learningRate*deltaout [ j] * hidd [l];
    bohi [ j] + = learningRate * deltaout [ j];
   }
  /* hidd -> in */
 for (int j = 0; j < nHiddenLayerNeurons; j++)
   for(int l = 0; l < nInputLayerNeurons; l++){
    deltain [ j] = 0;
     for (int k = 0; k < nOutputLayerNeurons; k++)
      deltain [ j] + = deltaout [k] * wohi [k] [j] * dF (neth [ j]);
    whin [ j] [l] + = learningRate * deltain [ j] * in [l];
    bhin [ j] + = learningRate * deltain [ j];
```

```
 }
 }

 public void decimalToBinary (int dec, int bin [], int n){
  int k = 0, r = 0, q = dec;
  while (q ! = 0){
    r = q%2;
   if (r == 0)
     r = –1;
    bin [k++] = r;
    q = q/2;
  }
    for (; k < n; k++)
   bin [k] = –1;
 }

 public void decimalToCode(int dec, int bin [], int n){
  for(int i = 0; i < n; i++)
    bin [i] = –1;
  if (dec < n)
    bin [dec] = 1;
 }

 public Integer predict (){
  int bin [] = new int [nNeuronsForSymbol];
  for(int i = 0; i < history; i++){
   decimalToBinary (((Integer)Q.get (i)).intValue (), bin, nNeuronsForSymbol);
   for (int j = 0; j < nNeuronsForSymbol; j++)
     input [i * nNeuronsForSymbol + j] = bin [ j];
  }
  forward (input);
  // finding the position of the maximum output which will be predicted
  int maxOutputPos = 0;
  for (int i = 1; i < nOutputLayerNeurons; i++)
   if (out [i] > out [maxOutputPos])
     maxOutputPos = i;
  return new Integer (maxOutputPos);
 }

 public void update (Integer observation){
  int tp [] = new int [nOutputLayerNeurons];
  Q.add (observation);
  If (Q.size () > history){
   Q.remove (0);
   decimalToCode (observation.intValue (), tp, nOutputLayerNeurons);
   backward (tp, input);
  }
 }
}
```

As it can be observed, the *predict* method codifies the input data (consisting in a certain history of observations) from decimal to binary, propagates the input forward through the network by calling the *forward* method and after that the index of the maximum output is considered as being the predicted observation. The *update* method adds the real observation symbol (when it is available) to the observation sequence, computes the errors existing between the real observation symbol and the predicted one and after that propagates these error terms backward through the network by calling the *backward* method. The goal of the backward step is to adjust the weights in order to minimize the error.

AbstractConfidence is another interface which relies on a concrete confidence class to define the *trust*, *untrust* and

isPredictable methods. The goal of the confidence mechanism is to decide, based on the current observation symbol or context and its attached confidence counter, if a potential prediction statistically likes to be correct or not. It dynamically classifies observation symbols or contexts into predictable and unpredictable and provides this classification through the *isPredictable* boolean method. The goal of the *trust* and *untrust* methods is to increase or decrease the confidence in a certain observation symbol or context when the prediction turns out to be correct or wrong, respectively.

```
public interface AbstractConfidence {
  public abstract void trust (Integer observation);
  public abstract void untrust (Integer observation);
  public abstract boolean isPredictable (Integer observation);
}
```

The *Confidence* class provides definitions for the *trust, untrust* and *isPredictable* methods. In this example the confidence mechanism is implemented based on a set of confidence counter with two predictable and two unpredictable states, but other variants are also possible. saturating counters, each one being associated to a distinct observation symbol. Figure 3 depicts a 4-state

The *Confidence* constructor receives as parameters the maximum number of distinct observations, the number of states and a threshold which is used by the *isPredictable* method. The current observation is classified as predictable only if its attached confidence is in a state higher or equal to the threshold's value.

The next code is a sequence from a client which generates statistics within a certain prediction process regarding

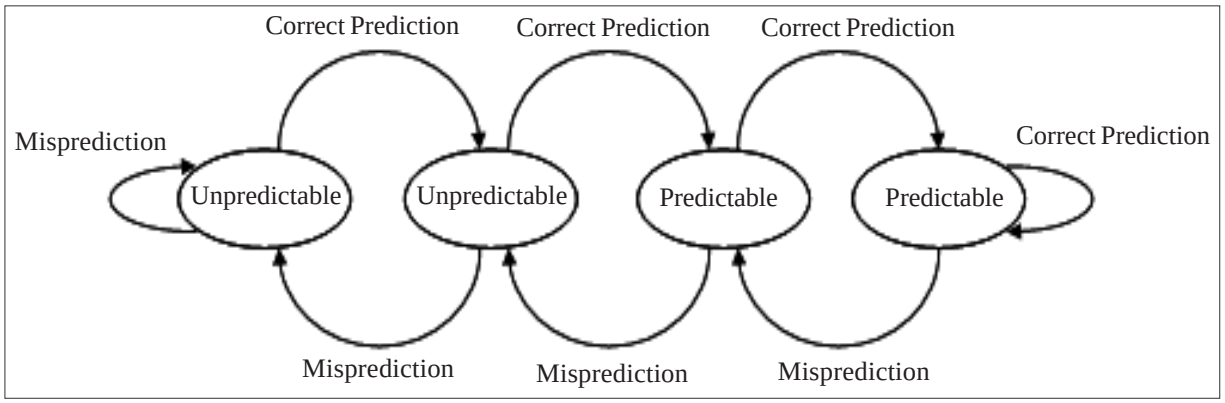

Figure 3. The confidence counter mechanism

The definition of the *Confidence* class is given below:

```
public class Confidence implements AbstractConfidence {
  int nStates;
  int threshold;
  int confidence [];

 public Confidence (int nObservationSymbols, int nStates, int threshold) {
    this.nStates = nStates;
    this.threshold = threshold;
   confidence = new int [nObservationSymbols];
  }

  public void trust (Integer observation){
   if (confidence [observation.intValue ()] < nStates –1 )
     confidence [observation.intValue ()] ++;
  }

  public void untrust (Integer observation){
   if (confidence [observation.intValue ()] > 0 )
     confidence [observation.intValue ()] —;
  }

  public boolean isPredictable (Integer observation){
   if (confidence [observation.intValue ()] > = threshold) return true;
   return false;
  }
}
```

```
void predictionProcess () throws java.io.IOException {
  Markov markov = new Markov (nObservationSymbols, order);
  Confidence confTable = new Confidence (nObservationSymbols, 4, 2);
  Integer current = null;
  String line = null;
  Integer next = null;
  int numberOfCorrectPredictions = 0;
  int numberOfPredictions = 0;
  double predictionAccuracy = 0.0;
  java.io.BufferedReader in = null;
  try{
   in = new java.io.BufferedReader (new java.io.FileReader ("benchmark.txt"));
  }
  Catch (java.io.FileNotFoundException fnfe){
   fnfe.printStackTrace();
  }
  line = in.readLine ();                          //reading the first observation
  markov.update (new Integer (line));
  current = new Integer (line);
  while ((line = in.readLine ()) != null) {
     next = new Integer (line);
     //checking predictability
     if (confTable.isPredictable (current))
       numberOfPredictions++;                     //total number of predictions
      //prediction of the next observation
     if (markov.predict ().equals (next)){       //correct prediction
       if (confTable.isPredictable (current))
         numberOfCorrectPredictions++;
        confTable.trust (current);
       }
       else confTable.untrust (current);          //misprediction
      markov.update (next);
      current = next;
  }
  in.close ();
  //computing prediction accuracy
  predictionAccuracy = numberOfCorrectPredictions / numberOfPredictions;
}
```

the number of predictions, the number of correct predictions and also the prediction accuracy (computed as the report between the number of correct predictions and the total number of predictions). First, it instantiates a *Markov* predictor and a *Confidence*. The observations are then read from a file called in this example *benchmark.txt*. For each current observation is determined if its attached confidence is in a predictable state or not, a prediction being performed or not, consequently. The predictor is updated with each new observation symbol. Thus, the *predictionProcess* method represents the kernel of the application which creates a bridge between the participant classes.

In the above example we used 4-state confidence counters with a threshold of 2 (meaning two unpredictable and two predictable states. Obviously, the confidence mechanism can be more selective, for example, having only one predictable state. In this case the threshold parameter of the *Confidence* constructor is 3 instead of 2.

A client which is using the MLP predictor is similar, but instead of a Markov object it instantiates a MLP object. It is also possible to attach a confidence counter to combinations of two [5] or more observation symbols instead of only one (as we did here).

4.2. Experimental results

The main goal of this work is to propose a design pattern for prediction algorithms, the evaluations provided in this section being just a validation example of the predictor design concept. The benchmark set used for the evaluations contains movement sequences of 4 employees in 14 rooms, acquired from the Smart Doorplates project developed at University of Augsburg [13]. Each file contains the location data of a single test person. The benchmarks are text files generated by recording the movements of the test persons through the offices located at the fourth

Original benchmark	Benchmark afterroom codification
2003.10.27 10:26:29;corridor;A;1067246789004	-
2003.10.27 10:26:35;402;A;1067246795003	0
2003.10.27 10:27:15;corridor;A;1067246835004	-
2003.10.27 10:27:20;412;A;1067246840003	1
2003.10.27 10:27:48;corridor;A;1067246868003	-
2003.10.27 10:27:51;402;A;1067246871659	0

Table 1. The first movements of person A before and after the room codification process

floor in the building of the Computer Science Institute at the University of Augsburg. Table 1 shows the contents of a benchmark before and after the room codification process.

Each line from the original benchmarks represents a person's movement, containing the movement's date and hour, the room's name, the person's name and a timestamp. After the codification process the benchmarks contain only the room codes (0÷13), because in this starting stage of our work only this information is used for prediction. In the codification process we have also eliminated from the benchmarks the common corridor, because it could behave as noise. There are two benchmark types: some short benchmarks containing about 300-400 movements and some long benchmarks containing about 1000 movements. Our evaluations are based on the long benchmarks.

For the experiments we used the best Markov predictor configuration obtained by [5] [15] having a learning rate of 0.1 a history of 2 rooms, $N = 8$ neurons in the input layer (4 neurons per room, enough to binary codify a maximum of 16 rooms) and 9 hidden layer neurons. The output layer contains one neuron for each room (14 neurons), the position of the highest output being considered the predicted room.

In this work we are interested to predict the next room from all rooms excepting the own office. Figure 4 presents comparatively the prediction accuracy (the report between the number of correct predictions and the total number of predictions, expressed as percentage) obtained using the Markov and MLP predictors without confidence and also with 4-state confidence counters having 1 predictable state (denoted $C1$) or 2 predictable states (denoted $C2$):

The best results were obtained using the MLP predictor with the $C1$ confidence. This method provided an average prediction accuracy of 84.5%, with a maximum of 93.58%. As Figure 4 shows, the confidence mechanism can increase the accuracy by avoiding prediction when the confidence in a certain context is low. Obviously, the confidence mechanism can missing from a predictor design, especially in applications where mispredictions do not affect the general performance, but is necessary in applications whose performances are decreased by mispredictions.

Table 2 shows how the prediction rate (the report between

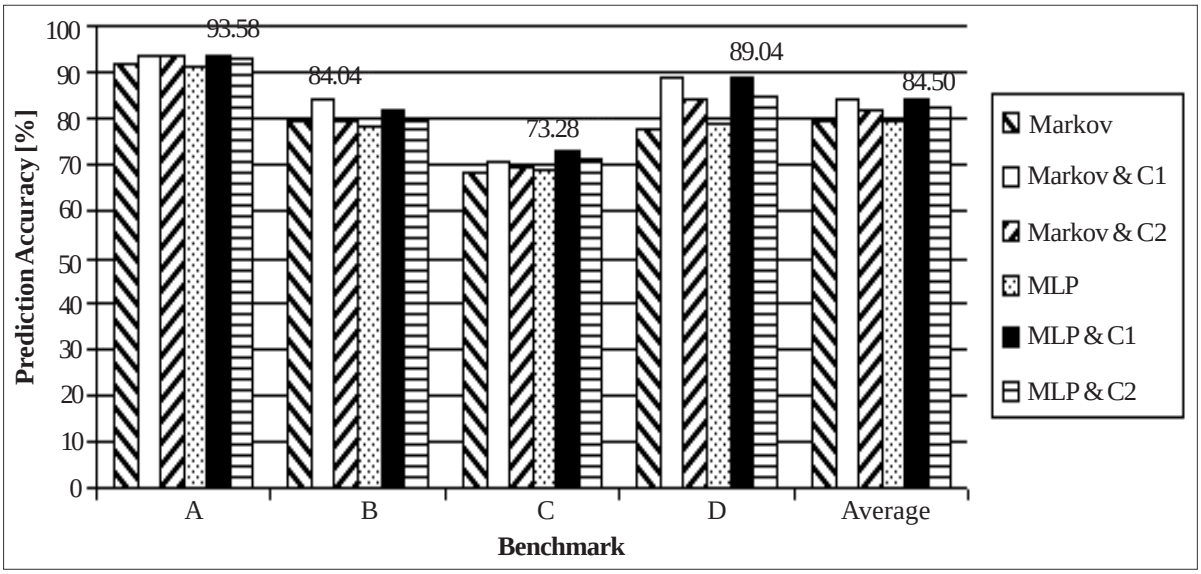

Figure 4. The prediction accuracies obtained using the Markov and MLP predictors, with and without confidence

Benchmark	Markov & C1	Markov & C2	MLP & C1	MLP & C2
A	70.27%	81.08%	70.27%	80.18%
B	61.05%	79.78%	60.67%	79.40%
C	50.57%	73.58%	49.43%	71.70%
D	60.67%	74.06%	61.09%	74.06%
Average	60.64%	77.12%	60.37%	76.33%

Table 2. The prediction rates obtained using the Markov and MLP predictors with C1 and C2 confidences

the number of predictions and the total number of movements, expressed as percentage) is influenced by the selectivity (threshold) of the confidence mechanisms for both predictors. The prediction rate of the predictors without confidence is 100% because a prediction is always performed. It can be observed that as more selective confidence mechanism we used as higher accuracy and lower prediction rate we obtained.

5. Conclusions

In this study, we have presented the Predictor design pattern. We described this new design pattern and provided an example which generates statistics regarding the prediction accuracy. Obviously, a client can provide timing measurements, too, such as we did for microarchitectural value predictors [7-9]. It is also possible for a certain client application to use hybrid predictors such as cascaded predictors or metapredictors, since usually a single predictor cannot capture all the types of predictability patterns. In the cascaded prediction approach multiple predictors are used in different stages, in a statically predefined order (fixed prioritization). A metapredictor uses multiple predictors in one stage and dynamically selects the best predictor (adaptive prioritization). A hybrid approach is motivated by the fact that even the MLP predictor provided the highest average prediction accuracy, on the B benchmark the Markov predictor was better.

Prediction is a widely used technique in computer science and engineering and thus, in our opinion, the proposed design pattern can be very useful for software developers but also for hardware architects, especially in designing the software simulators of microprocessors – an important stage of computer architecture research and design process [20].

References

[1] Aguilar, M., Barniv, Y., Garrett, A. (2003). Prediction of Pitch and Yaw Head Movements via Recurrent Neural Networks, *International Joint Conference on Neural Networks*, 4, p. 2813-2818.

[2] Freeman, E., Robson, E., Bates, B., Sierra, K. (2004). Head First Design Patterns, O'Reilly Media, USA.

[3] Gallant, S. I. (1993). Neural Networks and Expert Systems, *MIT Press*., USA.

[4] Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, USA.

[5] Gellert, A., Vintan, L. (2006). *Person Movement Prediction Using Hidden Markov Models*. *Studies in Informatics and Control*, National Institute for Research and Development in Informatics, 15 (1) 17-30.

[6] Gellert, A., Florea, A., Vintan, M., Egan, C., Vintan, L. (2007). Unbiased Branches: An Open Problem. Twelfth Asia-Pacific Computer Systems Architecture Conference (ACSAC'07), p. 16-27.

[7] Gellert, A., Florea, A., Vintan, L. (2009). Exploiting Selective Instruction Reuse and Value Prediction in a Superscalar Architecture. *Journal of Systems Architecture*, Elsevier, 55 (3) 188-195.

[8] Gellert, A., Palermo, G., Zaccaria, V., Florea, A., Vintan, L., Silvano, C. (2010). Energy-Performance Design Space Exploration in SMT Architectures Exploiting Selective Load Value Predictions. *International Conference on Design, Automation and Test in Europe* (DATE 2010), p. 271-274.

[9] Gellert, A., Calborean, H., Vintan, L., Florea, A. (2012). Multi-Objective Optimizations for a Superscalar Architecture with Selective Value Prediction. *IET Computers & Digital Techniques*, 6 (4) 205-213, (July).

[10] Kerievsky, J. (2004). Refactoring to Patterns, Addison-Wesley, USA.

[11] Mitchell, T. (1997). Machine Learning, McGraw-Hill, USA.

[12] Mozer, M. C. (2004). Lessons from an adaptive house, Smart Environments: *Technology, Protocols, and Applications*, J. Wiley & Sons, USA.

[13] Petzold, J. (2004). Augsburg Indoor Location Tracking Benchmarks. Technical Report 2004-9, *Institute of Computer Science*, University of Augsburg, Germany.

[14] Rabiner, L. R. (1989). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition, *In*: Proceedings of the IEEE, 77 (2) 257-286.

[15] Vintan, L., Gellert, A., Petzold, J., Ungerer, T. (2004). Person Movement Prediction Using Neural Networks. *In*: Proceedings of the KI2004 International Workshop on Modeling and Retrieval of Context (MRC 2004), 114, p. 618-623.

[16] Vintan, L., Florea, A., Gellert, A. (2005). Focalising Dynamic Value Prediction to CPU's Context. *IEE Proceedings – Computers & Digital Techniques*, 152 (4) 457-536.

[17] Vintan, L., Gellert, A., Florea, A., Oancea, M., Egan, C. (2006). Understanding Prediction Limits through Unbiased Branches. *Eleventh Asia-Pacific Computer Systems Architecture Conference* (*ACSAC*'06), p. 483-489.

[18] Vintan, L., Florea, A., Gellert, A. (2008). Random Degrees of Unbiased Branches. *In*: Proceedings of the Romanian Academy, Series A, (3) 259-268.

[19] Vlissides, J. (1998). Pattern Hatching: Design Patterns Applied, Addison-Wesley, USA.

[20] Yi, J. J., Lilja, D. J. (2006). Simulation of Computer Architectures: Simulators, Benchmarks, Methodologies and Recommendations. *IEEE Transactions on Computers*, 55 (3) 268-280.