*Journal of the Chinese*
*Statistical Association*
*Vol. 48, (2010) 190–226*

# DESIGN PATTERNS FOR STATISTICAL AND GRAPHICAL ANALYSIS

**Wen Hsiang Wei**

**Department of Statistics, Tung Hai University**

**ABSTRACT**

Design patterns, a common discussion topic in software development teams in the world and one of the techniques to support software reuse, is used to develop statistical packages `JavaStat` and `StatGraphics`. Several well-known patterns used to implement statistical methods or algorithms are introduced. In addition, based on these patterns, a new pattern, referred to as Data Analysis, is proposed to help the statisticians develop code for data analysis. The applications of the proposed pattern are demonstrated via several examples. In addition, how design patterns can provide a solution deployed in R is also discussed via illustrative examples.

Key words and phrases: Design Patterns, Interface, Data Analysis Pattern, R, Statistical Software, UML.
JEL classification: C63

## 1. Introduction

The development of statistical computing environments is an exciting area. Many statisticians around the world are using statistical packages to reach at conclusions. Developing useful and modern statistical packages is thus crucial.

One of the commonly used software engineering strategies is software reuse in the design process. Reuse-based development is now a mainstream approach to software engineering (Sommerville, 2007, p.390). One of the techniques for software reuse is design

patterns. Design patterns, originally used in civil engineering and architecture (Alexander et al., 1977) and popularized by GoF (the Gang of Four, Gamma et al., 1995) in computer science, is a common discussion topic in software development teams around the world. GoF introduced twenty-three design patterns, which experienced programmers have used to design object-oriented software. These patterns enable programmers to create extensible, powerful, elegant, and most importantly, reusable designs in software development process. Several advantages of using design patterns are reducing risk, saving time and energy, and improving the programmers' skill and apprehension (Tate, 2002, pp.7–8). In the classical book by Fowler et al., (1999), several patterns were adopted for code refactoring (see Fowler et al., 1999, Chapters 8, 10 and 11), which is another commonly used software engineering technique and can improve the original code without changing its external behavior. Design patterns have been extensively used in Java APIs (Application Programming Interface). Furthermore, commonly used design patterns can be also used for implementing statistical methods or algorithms. Therefore, the first goal of this paper is to describe how design patterns can be used to implement statistical methods or algorithms efficiently and effectively. These patterns implemented with sample code are introduced in Section 2.

A design pattern can be described using a specified format (see GoF, pp.6–8). Several important sections of a patterns are the name of the pattern, the context in which it arises, the motivation (or forces), the structure, and the examples along with the sample code. A new pattern, referred to as Data Analysis, is proposed to help the statisticians develop code for statistical analysis. Two packages `JavaStat` and `StatGraphics`, consisting of basic modules that provide the user-level functionality for basic statistics, are based on the pattern. Therefore, the second goal of this paper is to introduce the new pattern and illustrate how the new pattern can be used to develop the classes for statistical and graphical analysis. The proposed design pattern has been incorporated with the framework "Data Analysis Module" proposed by Wei and Chen (2008). As indicated by several statisticians, Java is a very useful language for the implementation of a statistical computing environment (see Chambers, 1999, 2000; Krätzig, 2007; Liang and Huang, 2009; Warnes, 2002; West et al., 1998; West et al., 2004). Therefore, the two packages were implemented using Java. The Java code

in the two packages is currently licensed under the terms of the GNU (General Public License). The two packages can be downloaded from the site

$$https://sourceforge.net/projects/javastat/$$

To present the proposed pattern in a nutshell, the approach of GoF is adopted in this article. The overall structure is as follows.

- Name: Data analysis.
- Intent: Define the required classes for data analysis and establish the relationship among these classes.
- Motivation: In general, for data analysis, there are four key parts, including the data of interest, the arguments for specifying some attributes related to the analysis, the procedure for obtaining the required quantities, and the output for these quantities. The data of interest, arguments, and output are static. On the other hand, the procedure is dynamic. By suitably arranging classes for the four parts, a guideline can be given for the fitting process.
- Structure and participants: See Section 3.1.
- Implementation and sample code: See Section 3.2.
- Examples: The pattern can be used to construct new classes for statistical methods, which is the ultimate goal of the paper. Two examples are presented in Section 3.3.
- Related patterns: The ones are introduced in Section 2.

Although the sample code for the proposed design pattern and its related patterns are implemented with Java, design patterns can be implemented in other object-oriented programming languages, for example C++. In addition, design patterns can fit in with the rather different but widely used language R. A discussion on how design patterns fit in with R is given in Section 4. Examples are also provided for illustrations. Finally, a concluding discussion is given in Section 5.

## 2. GOF Design Patterns

Three patterns, Template Method, Strategy, and Factory Method, are introduced along with sample code in the following sub-sections. These patterns are related to the proposed pattern and can be used for refactoring. The complete code can be found in the directory `examples`.

### 2.1 Template method

Template Method, framing the skeleton of an application (GoF, pp.325–330), can be used to implement various parts of a statistical analysis, depending on whether the problem of interest can be implemented by the sub-classes without compromising the overall structure of the statistical analysis. For example, suppose that a statistician wants to implement a general regression analysis for obtaining the statistics of interest, presenting the results, and generating the required plots. The abstract class `AnalysisTemplate` defines the required methods and operation. This class plays the role of the class `AbstractClass` in the Template Method.

```
public abstract class AnalysisTemplate{
  public Object statistics;
  public abstract Object getStatistics(double[] response,
    double[][] covariate);
  public abstract void output(Object statistics);
  public abstract void getPlot(Object statistics);
  public void doAnalysis(...){
(a) statistics = getStatistics(response, covariate);
(b) output(statistics);
(c) createPlot(statistics);
  }
}
```

The code with the letters in the method *doAnalysis* is symbolic of the steps for doing regression analysis. The method *doAnalysis* plays the role of the method *Tem-*

*plateMethod*, while the three abstract methods *getStatistics*, *output*, and *createPlot* play the role of the method *PrimitiveOperation*. The statistic is assigned to the data member `statistics`, while the input responses and covariates are assigned to the data members `response` and `covariate`, respectively. Suppose the statistician wants to perform both a linear regression analysis and a logistic regression analysis, the different output results for these analyses can be displayed or saved in different ways. The plots to be generated for these analyses are also different. These variant parts can be implemented by the following subclasses `LinearRegressionAnalysis` and `LogisticRegressionAnalysis`.

```
public class LinearRegressionAnalysis
  extends AnalysisTemplate{
  public LinearRegression getsStatistics(double[] response,
    double[][] covariate){
    return statistics = new LinearRegression(
      response, covariate);
  }
  public void output(Object statistics){
    /* Saves the result in a file */
              ...
  }
  public void createPlot(Object statistics){
    /* Creates two plots */
              ...
  }
}

public class LogisticRegressionAnalysis
  extends AnalysisTemplate{
  public LogisticRegression getsStatistics(
    double[] response, double[][] covariate){
    return statistics = new LogisticRegression(
```

```
      response, covariate);
  }
  public void output(Object statistics){
    /* Displays the result on the terminal */
              ...
  }
  public void createPlot(Object statistics){
    /* Creates a residual plot */
              ...
  }
}
```

The two subclasses play the role of the class `ConcreteClass` in the pattern. Note that the method *doAnalysis* needs not be implemented in these subclasses and thus code duplication can be avoided. That is, the class `AnalysisTemplate` provides a guideline for the statistician to create new classes for similar statistical analyses. Since the classes for the statistics of interest is highly variable, the class for the argument in the methods *output* and *createPlot* is `Object`, which is the parent class or superclass of all Java classes. In addition, since the ways to manage the results and the plots to be generated are all different, the approach by creating an interface defining the common quantities and then extracting them in the class implementing the interface might not work for this example.

## 2.2 Strategy

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable (GoF, pp.315–323). Thus, the implementation of the classes for different statistical methods (algorithms) can be created based on the Strategy pattern. It is very similar to the use of interfaces, which permits users to replace existing components of some algorithm with versions tuned to the specific problem, as indicated by Warnes (2002, p.5). Suppose that a statistician wants to conduct a simulation study to compare the performance of the estimators of the location and

dispersion parameters. These estimators include the commonly used sample mean, variance, median, and median absolute deviation (MAD). The statistician could implement a family of algorithms to obtain these estimators. In the following code, the interface `EstimationStrategy`, playing the role of the interface `Strategy`, defines the required methods *locationEstimate* and *dispersionEstimate* common to the estimators of interest. The two methods play the role of *AlgorithmInterface*. The two classes `UnbiasedEstimate` and `RobustEstimate`, playing the role of the class `ConcreteStrategy` in the pattern, implement the methods *locationEstimate* and *dispersionEstimate* for computing the sample mean, variance, median, and MAD, respectively. The input data can be assigned to the data member `data`.

```
public interface EstimationStrategy {
  double locationEstimate(double[] data);
  double dispersionEstimate(double[] data);
}

public class UnbiasedEstimate implements EstimationStrategy{
  /* Calculates the sample mean */
  public double locationEstimate(double[] data){...}
  /* Calculates the sample variance */
  public double dispersionEstimate(double[] data){...}
}

public class RobustEstimate implements EstimationStrategy{
  /* Calculates the sample median */
  public double locationEstimate(double[] data){...}
  /* Calculates the sample median absolute deviation */
  public double dispersionEstimate(double[] data){...}
}
```

Like the Template Method pattern, one of the advantages for using the Strategy pattern is to factor out common operations of the family of statistical methods (algo-

rithms). Further, to obtain the mean square error of the estimator of interest, the code is the following.

```
public double[] meanSquareError(double[] trueValue,
  double[][] data, EstimationStrategy strategy){
            ...
  for(int i=0; i < data.length; i++){
    meanSquareError[0] += Math.pow(strategy.
      locationEstimate(data[i]) - trueValue[0], 2.0);
    meanSquareError[1] += Math.pow(strategy.
      dispersionEstimate(data[i]) - trueValue[1], 2.0);
  }
            ...
}
```

The method *meanSquareError*, playing the role of *ContextInterface*, can be applied to all classes implementing the interface `EstimationStrategy`. The input arguments include `trueValue` being the true values of the parameters of interest, `data` being the data, and `strategy` being the estimator. The required mean square error is assigned to the data member `meanSquareError`. The statistician needs not create two different methods for computing the mean square error for different estimators. The statistician can thus use the family of classes by only referring to the interface `Strategy`, which is played by the interface `EstimationStrategy` in the example. Note that the method *meanSquareError* can be also applied for new classes, which will be created in the future and implement the interface `EstimationStrategy`. The other advantage of using the pattern is the connection with the dynamic loading feature in Java. Dynamic loading allows a computer program to startup in the absence of some classes, to search for available classes, and then to extend additional functionality. For example, in JDBC (Java DataBase Connectivity) API (Application Programming Interface), loading for the database driver is dynamic (Reese, 2000). In this statistical example, suppose the statistician just gets a compiled class `WinsorizedEstimate`, which implements the interface `EstimationStrategy` and can be used for computing

the Winsorized location and dispersion estimates, then, the statistician can dynamically load the class to obtain the robust estimates without modifying the original code, as illustrated by the following code. The class to be loaded can be specified by the argument `className`. The compiled class `WinsorizedEstimate` can be loaded and executed at runtime by the MS-DOS (Microsoft-Disk Operating System) command "java examples.strategy.StrategyExample examples.strategy.WinsorizedEstimate". The object `loader` can be used to load the required class. Then, the loaded class is specified by `statClass`, instantiated by the object `obj`, and then assigned to the object `strategy` of which class is `EstimationStrategy`. Finally, the location and dispersion estimates can be obtained and assigned to the data members `locationEstimate` and `dispersionEstimate`, respectively. Note that all the classes implement the interface `EstimationStrategy` can be loaded dynamically.

```
public class StrategyExample{
           ...
  public static void main(String[] className)
    throws Exception{
    StrategyExample example = new StrategyExample();
             ...
    /* Initializes the class loader of StrategyExample */
    ClassLoader loader =
      example.getClass().getClassLoader();
    /* Loads the class className[0] dynamically */
    Class statClass = loader.loadClass(className[0]);
    Object obj = statClass.newInstance();
    EstimationStrategy strategy = (EstimationStrategy) obj;
    double locationEstimate =
      strategy.locationEstimate(...);
    double dispersionEstimate =
      strategy.dispersionEstimate(...);
             ...
```

```
  }
}
```

## 2.3 Factory method

The Factory Method pattern (GoF, pp.107–116) defines an interface for creating a required object, but lets subclasses instantiate the required class. The pattern is very suitable for creating common objects obtained by different statistical methods, for example, the plots or output reports. Suppose a statistician wants to save the output from a statistical analysis in some files, the output could be a plot, a table, or simply a numeric value. Thus, these different outputs can be considered as "products" created by different factories, such as the factory producing the plots or the one producing the tables. The following code illustrates the use of the pattern.

```
public abstract class OutputCreator{
  public abstract Object output(...);
  public void outputSerialized(...){
             ...
    /*   Saves the output in a file */
    outfile.writeObject(output(...));
  }
           ...
}

public class DoubleArrayOutputCreator extends OutputCreator{
  /*   Generates the numeric output */
  public double[] output(...){...}
           ...
}

public class FigureOutputCreator extends OutputCreator{
  /*   Generates the figure object output */
```

```
  public JFreeChart output(...){...}
          ...
}
```

The class `OutputCreator` plays the role of the class `Creator` in the pattern, while the two subclasses `DoubleArrayOutputCreator` and `FigureOutputCreator` play the role of the class `ConcreteCreator`. The method *output*, playing the role of the method "*FactoryMethod"*, produces the output. The different outputs play the role of the class `ConcreteProduct`. In addition, the method *outputSerialized* plays the role of the method "*AnOperation*". The method calls the factory method *output* to create the output (`Product`) and saves it in a file by object serialization, which is a storage mechanism for objects in Java and can ensure the programmers not to worry about details of file formats and input/output. The advantage of using the pattern is to prevent the bind with specific classes. In the above example, rather than create two *outputSerialized* methods in the subclasses, the statistician can save different outputs in some files by using the method in the class `OutputCreator`. Further, the new output such as a two-dimensional double array can be created by adding the associated method in the class `DoubleArrayOutputCreator` instead of creating a new class.

## 3. Data Analysis Pattern

### 3.1 Structure and participants

Based on the patterns introduced in the previous section, a design pattern, referred to as Data Analysis, is proposed to help the statisticians develop code for data analysis. The unified modeling language (UML) diagram for the pattern is given in Figure 1. Note that multiple methods can be implemented in a class, as indicated by the dots below the method in the figure. The class `StatisticalAnalysis` defines the required components or methods in data analysis, as given by the dashed lines pointing to three classes `ObjectArgument`, `ObjectData`, and `ObjectOutput`. The class `StatisticalAlgorithm` defines and implements the various operations for the data analysis. This class could be the class `AbstractTemplate` in the Template Method,

or the interface `Strategy` in the Strategy pattern, or the class `Creator` in the Factory Method. The classes `MethodA`, `MethodB`, and `MethodC` implement the methods defined in the class `StatisticalAlgorithm`. Therefore, these classes can be the class `ConcreteClass` in the Template method, or the class `ConcreteStrategy` in the Strategy pattern, or the class `ConcreteCreator` in the Factory Method. Since some statisticians may be used to the argument with primitive data type and the others to the one with object data type, the dashed lines thus point to the classes `PrimitiveArgument` and `ObjectArgument` from the classes `MethodA`, `MethodB`, `MethodC`, and `StatisticalAlgorithm`. The class `ObjectArgument` can be an enumeration. Then, some methods,

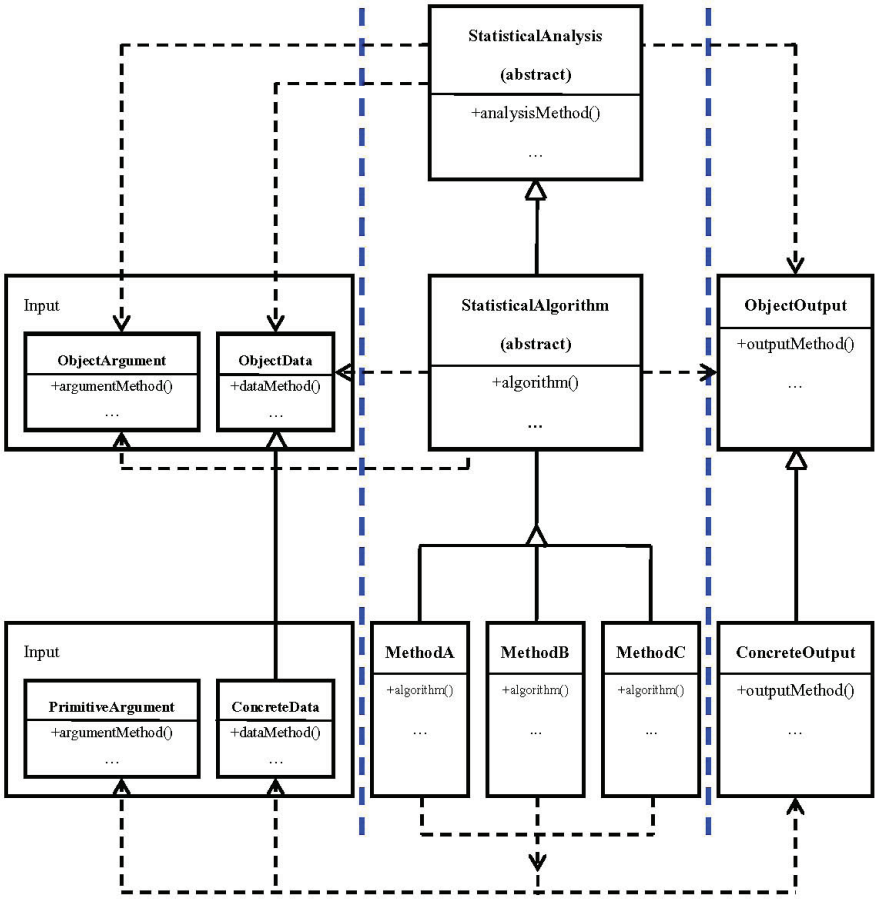


Figure 1 Data Analysis Class Relationships

playing the role of argumentMethod, can be used to describe these arguments and in which classes they are used. Similarly, the type of the input data can be very general such as `Object[]` or specific as `int[]`. It is indicated by the dashed lines pointing to the classes `ObjectData` and `ConcreteData`, respectively. The abstraction of the input data can help the programmers reduce implementation dependencies between sub-classes and thus develop methods applicable to different statistical problems. Finally, for the output with general or specific types, the classes `ObjectOutput` and `ConcreteOutput` can be created, as indicated by the dashed lines pointing to them from the classes `StatisticalAlgorithm`, `MethodA`, `MethodB`, and `MethodC`. The classes `MethodA`, `MethodB`, and `MethodC` can also point to the class `StatisticalAnalysis` directly. One application, such as the one in Section 3.3.1, is to construct the classes for general purposes. In such case, these classes only use some classes implementing the abstract methods defined in the class `StatisticalAlgorithm`.

## 3.2 Implementation and sample code

The classes in the packages `JavaStat` and `StatGraphics` are based on the proposed pattern. The classes `StatisticalAnalysis` and `GraphicalAnalysis` in the two packages play the role of the abstract class `StatisticalAnalysis` in Figure 1. The code for the two classes is as follows.

```
public abstract class StatisticalAnalysis{
  public Hashtable argument = new Hashtable();
  public Object[] dataObject;
  public Hashtable output = new Hashtable();
           ...
  public Hashtable getOutput(){ return this.output; }
}

public abstract class GraphicalAnalysis{
  public Hashtable argument = new Hashtable();
  public Object[] dataObject;
```

```
  public Hashtable output = new Hashtable();
  public JFreeChart plot;
          ...
  public JFreeChart getPlot(){ return this.plot; }
}
```

The data of interest, input arguments, and output can be specified via the data members `argument`, `dataObject`, and `output`. In addition, for graphical analysis, the generated plot can be also assigned to the data member `plot`. The two default methods *getOutput* and *getPlot* for obtaining the output in data analysis and generated plot are defined and named *analysisMethod*. To implement the classes `StatisticalAlgorithm`, `MethodA`, `MethodB`, and `MethodC`, the following sub-sections provide examples for a variety of statistical problems. The complete code can be found in the directory `src`. Note that some functions in the examples were named as nouns rather than verbs. It is similar to the *sin* and *cos* methods of the class `java.lang.Math` having mathematically conventional names (Gosling et al., 2005, p.146). Also, it is similar to the naming conventions of some statistical software, such as Splus or R.

### 3.2.1 Statistical inference for one and two-sample problems

Suppose the parameter $\theta$ is of interest, such as the population mean and proportion in one-sample problems, or the mean difference and proportion difference in two-sample problems, the common quantities of interest are confidence interval, test statistic, and the associated *p-value*. The statistician could implement these methods in the face of different statistical problems. However, code duplication can not be avoided and some generated classes are very similar. Further, for a new class with similar methods, the statistician needs to implement these various parts in a duplicate way. Another disadvantage of such implementations is the complexity for code maintenance. Suppose the formula for the *p-value* is wrong, then the statistician has to debug all classes with the methods for computing the *p-value*, or even worse to check other classes using these methods. The Data Analysis pattern can be used to avoid code duplication and help the statistician to develop structured and succinct classes. Further, the resulting classes

are easy to maintain.

In general, under the assumption that the null distribution of the test statistic is symmetric, the confidence interval, test statistic, and the associated *p-value* can be expressed by

$$\hat{\theta} \pm c_\alpha s_{\hat{\theta}}, (\hat{\theta} - \theta_0)/s^*_{\hat{\theta}}, 2i_1[1 - max(F, 1 - F) + i_2 F + (1 - i_1 - i_2)(1 - F)],$$

where $\hat{\theta}$ is the point estimate, $c_\alpha$ is the critical value at a significance level of $\alpha$, $\theta_0$ is the null value of $\theta$, $s_{\hat{\theta}}$ is the standard error of the point estimate, $s^*_{\hat{\theta}}$ is the standard error of the point estimate as the null hypothesis is true, $F$ is the cumulative null distribution function evaluated at the value of the test statistic, and two indexes, $i_1$ and $i_2$, are associated with the specification of the alternative hypothesis. The above equations hold for a variety of inference problems, including one- and two-sample problems for population means and proportions. Below is the abstract class `StatisticalInferenceTemplate`, which defines the required methods associated with these statistical quantities and how these quantities can be obtained. This class can provide the class `StatisticalAlgorithm` in the Data Analysis pattern.

```
public abstract class StatisticalInferenceTemplate
  extends StatisticalAnalysis{
  public double pointEstimate;
  public double pointEstimateSE;
  public double nullValue;
  public double [] confidenceInterval;
  public double testStatistic;
  public double pValue;
  public abstract Object pointEstimate(...);
  public abstract Object pointEstimateSE(...);
           ...
  public Object confidenceInterval(...){
             ...
    confidenceInterval = new double[]{
```

```
      pointEstimate - criticalValue * pointEstimateSE,
      pointEstimate + criticalValue * pointEstimateSE};
              ...
    return confidenceInterval;
  }
  public Object testStatistic(...){
              ...
    testStatistic =
      (pointEstimate - nullValue) / pointEstimateSE;
              ...
    return testStatistic;
  }
  public double pValue(double[] sideIndex, double cdf){
              ...
    pValue = 2 * sideIndex[0] *
      (1 - Math.max(cdf, 1.0 - cdf)) + sideIndex[1] * cdf +
      (1.0 - sideIndex[0] - sideIndex[1]) * (1.0 - cdf);
              ...
    return pValue;
  }
}
```

For computing the point estimates and their standard errors in the one-sample problem, the abstract methods can be implemented by the two sub-classes of the abstract class `StatisticalInferenceTemplate`, `OneSampMeanTest` and `OneSampProp`. The methods *pointEstimate* in the classes `OneSampMeanTest` and `OneSampProp` can be used to calculate the sample mean and proportion, respectively, while the methods *pointEstimateSE* to obtain the standard errors of the two point estimates. Similarly, for the two-sample means and proportions problems, the methods for obtaining required point estimates and their standard errors are given in the classes `TwoSampMeansTest` and `TwoSampProps`, respectively. In the class `StatisticalInferenceTemplate`, the

common operations for obtaining the required quantities have been implemented by the methods *confidenceInterval*, *testStatistic*, and *p-value* and assigned to the data members with the same names. The point estimate $\hat{\theta}$, its associated standard errors $s_{\hat{\theta}}$ or $s^*_{\hat{\theta}}$, and the critical value $c_\alpha$ are assigned to the data members `pointEstimate`, `pointEstimateSE`, and `criticalValue`, respectively. The data member `nullValue` represents the null value $\theta_0$, `cdf` represents the value of $F$, and `sideIndex` represents the vector of the indexes $i_1$ and $i_2$. On the other hand, the different operations for the one- and two-sample problems are delegated to the sub-classes. To compute these quantities, the methods *confidenceInterval*, *testStatistic*, and *p-value* of the class `StatisticalInferenceTemplate` can be accessed by these sub-classes. Further, if the general formulae for computing the required quantities need to be modified, only the code in the class `StatisticalInferenceTemplate` needs to be changed.

### 3.2.2 One way ANOVA and tests for independence

The common statistical quantities for a multiple-sample problem are the test statistic and its associated *p-value*. Thus, two required methods for obtaining the two quantities are defined in the class `StatisticalInference`, which plays the role of the class `StatisticalAlgorithm`.

```
public abstract class StatisticalInference
  extends StatisticalAnalysis{
  public abstract Object testStatistic(...);
  public abstract Object pValue(...);
}
```

Then, the classes `OneWayANOVA` and `ChisqTest`, implementing the two required methods, play the role of the class `Method` in the Data Analysis pattern.

### 3.2.3 IRLS (iterative reweighted least squares) in generalized linear models

Consider the standard generalized linear model (Mccullagh and Nelder, 1989) in

which each component of the response vector has a distribution taking the form

$$f(y_i;\theta_i,\phi) \;=\; \exp\left\{\frac{[y_i\theta_i - b(\theta_i)]}{a(\phi)} + c(y_i,\phi)\right\},\; i=1,\ldots,n,$$

where $\theta_i$ and $\phi$ are scalar parameters, and $a(\cdot)$, $b(\cdot)$ and $c(\cdot)$ are specific functions. The dependence of the response $y_i$ on the associated explanatory variables $\boldsymbol{x}_i$ can be modeled through the link function $g(\cdot)$, where $g(\mu_i) = \eta_i = \boldsymbol{\beta}^t(\boldsymbol{x}_i), \mu_i$ is the mean of the response, and $\boldsymbol{\beta}$ are some parameters. The natural link and $a(\phi) = 1$ are assumed hereafter. Therefore, $\eta_i = \theta_i$.

The estimate of the coefficient vector $\boldsymbol{\beta}$ at the $(t+1)$'th iteration can be written as a weighted least squares estimate,

$$\hat{\boldsymbol{\beta}}_{t+1} = \left[\boldsymbol{X}^t\boldsymbol{W}(\hat{\boldsymbol{\beta}}_t)\boldsymbol{X}\right]^{-1}\boldsymbol{X}^t\boldsymbol{W}(\hat{\boldsymbol{\beta}}_t)\boldsymbol{z}(\hat{\boldsymbol{\beta}}_t),$$

where $\boldsymbol{X}$ is the covariate matrix and both the $n \times n$ weight matrix $\boldsymbol{W}(\hat{\beta}_t)$ and $n \times 1$ vector $\boldsymbol{z}(\hat{\beta}_t)$ depend on the distribution of the response and the link function. The estimate of the coefficient vector can be obtained by iteratedly computing the weighted least-squares estimate. The IRLS algorithm can be applied to diverse types of data. As applying the IRLS algorithm, two basic components, the elements of the weight matrix and the mean of the response, could vary with different types of data but the steps in obtaining the estimate of the coefficient vector are invariant. Thus, the various steps of the IRLS algorithm can be implemented by a class `GLMTemplate`, which plays the role of the class `StatisticalAlgorithm` in the Data Analysis pattern and is shown below. On the other hand, the model-dependent steps for different models, such as logistic regression model or log-linear model , can be delegated to the sub-classes. In the class `GLMTemplate`, the IRLS algorithm has been implemented by the method *coefficients*. The basic components are assigned to the data members `weights` and `means`, respectively, while the IRLS estimate is assigned to the data member `coefficients`. Note that the methods for computing the weight matrix and the mean of the response are abstract and thus need to be implemented by the sub-classes. Two sub-classes of the class `GLMTemplate`, `LogisticRegression` and `LogLinearRegression`, for fitting a logistic regression model and a log-linear model, respectively, then implement the model-dependent methods *weights* and *means*. The two classes play the role of the class `Method` in the Data Analysis pattern.

```
public abstract class GLMTemplate
  extends StatisticalAnalysis{
  public double[][] weights;
  public double[] means;
  public double[] coefficients;
  protected double[] coefficients(...){
              ...
        weights = weights(...);
              ...
        means = means(...);
              ...
        return coefficients;
  }
            ...
  protected abstract double[][] weights(...);
  protected abstract double[] means(...);
            ...
}
```

### 3.2.4 Testing survival curve differences

A general class of tests for the comparison of survival curves can be written as $\sum w_i r_i$, where $r_i$ is the difference between the number of deaths and its expected value in one group at the $i$'th ordered time and $w_i$ is the weight function. The variances of the class of tests can be written as $\sum w_i^2 V_i$, where $\sum V_i$ is the variance of the statistic $\sum r_i$. Different tests can be obtained by employing different weight functions, for example, the log-rank test as $w_i = 1$. Thus, the class `SurvivalTestTemplate`, playing the role of the class `StatisticalAlgorithm` in the Data Analysis pattern, defines the required methods and implements the common steps for the class of tests. Since the weight function $w_i$ is test-dependent, the associated method *weight* with the input argument `parameter` is abstract and needs to be implemented by the subclasses. On

the other hand, the method *testStatistic* calculates the test statistic and its associated variance and then assigned them to the data members `testStatistic` and `variance`, respectively. Since the calculations of the test statistics for different tests are similar, these invariant steps are implemented by the method *testStatistic*. The test-dependent weight functions for different tests can be implemented by the subclasses `LogRankTest` and `WilcoxonTest`, which play the role of the class `Method` in the proposed pattern. The code for these classes is given below.

```
public abstract class SurvivalTestTemplate
  extends StatisticalAnalysis
  implements SurvivalTwoSampTestsInterface{
  public abstract double weight(double parameter);
  public double testStatistic(...){
              ...
    testStatistic + = weight(...) * ...;
    variance + = Math.pow(weight(...), 2.0) * ...;
              ...
  }
            ...
}

public class LogRankTest extends SurvivalTestTemplate{
            ...
  public double weight(double parameter){ return 1.0; }
            ...
}

public class WilcoxonTest extends SurvivalTestTemplate{
            ...
  public double weight(double parameter){
    return parameter;
```

```
  }
          ...
}
```

### 3.2.5 Generating statistical plots

The class `PlotFactory` defines the method *createPlot* for generating the plots. Two classes in the package `StatGraphics`, `Plot2DFactory` and `Plot3DFactory`, implement the method *createPlot* for generating a variety of plots. The two classes play the role of the class `Method` in the proposed pattern.

## 3.3 Examples

In this section, two examples implementing Java classes are given to illustrate the effectiveness and reusability of the proposed pattern.

### 3.3.1 General statistical tests, regression analyses and statistical plots

Design patterns is good at organizing the classes in a systematical way and so is the Data Analysis pattern. Based on the Data Analysis pattern, the programmers can create a class, which can automatically invoke the required statistical methods based on the input data. In the following example, suppose that the users want to compare the proportions of two populations and means of three populations. The first five lines specify the required packages used in the class `StatisticalTestsExample` by the key word *import*. The data were taken from the textbook for basic statistics by Anderson, Sweeney and Williams (2001, p.408; p.434). The data of the quality-awareness examination scores for the employees in three different plants (populations) were assigned to the data member `anovaData`. In the other case, the numbers of tax returns with errors from two offices were 35 and 27, while the total numbers of tax returns were 250 and 300. By simply providing the input data, the users can conduct the two statistical tests using the class `StatisticalTests`. These results were assigned to the data members `statObj1` and `statObj2`. The results obtained were quite consistent with the ones given in the book.

```
import static java.lang.System.out;
import java.util.*;
import javastat.*;
import static javastat.util.Argument.*;
import javastat.util.*;
public class StatisticalTestsExample
{
  public static void main(String[] args)
  {
    double [][] anovaData = {{85, 75, 82, 76, 71, 85},
      {71, 75, 73, 74, 69, 82}, {59, 64, 62, 69, 75, 67}};
    Hashtable argument = new Hashtable();

    /* One-way ANOVA */
    StatisticalAnalysis statObj1 = new StatisticalTests(
      argument, anovaData).statisticalAnalysis;
    /* Two-sample proportions z test */
    StatisticalAnalysis statObj2 = new StatisticalTests(
      argument, 35, 250, 27, 300).statisticalAnalysis;
    out.println("\n" + statObj1.output.toString());
    out.println("\n" + statObj2.output.toString());
  }
}

Output:
One way ANOVA:
{TEST_STATISTIC=9.0,..., PVALUE=0.002702899474476883}

Two-sample proportions:
{TEST_STATISTIC=1.846189280616294,..., PVALUE=0.0648647268570739}
```

To conduct other tests, the users can use the data member `argument` to specify the tests of interest. The name of the test can be specified by the statement `argument.put(TEST_TYPE, TEST NAME)`, for example, `argument.put(TEST_TYPE, "RankSum")` for Wilcoxon rank sum test. Note that the users need not worry the order of the required arguments to obtain the required objects. Totally 14 different tests can be performed by the class `StatisticalTests`. Similarly, the class `RegressionModels` can be used for accommodating different regression models, as illustrated by the following code. The data were taken from the books by Anderson, Sweeney, and Williams (2001, p.468) and Collett (1994, pp.290–291). The quarterly sales of 10 Armand's Pizza Parlor restaurants located near college campuses were assigned to the data member `response`, while the sizes of the student population to the data member `covariate`. A linear regression analyses to reveal the relationship between quarterly sales and student population can be carried out by using the class `RegressionModels`. The other example was the study of different chemotherapy treatment of ovarian cancer. The survival times of 26 patients were assigned to the data member `survivalTime`, while the data member `survivalCensor` was the censor indicator. Two factors, the treatment and the age, were of interest and were specified by the data member `survivalCovariate`. A Cox proportional hazards regression model was fitted by using the class `RegressionModels`. These results were assigned to the data members `statObj1` and `statObj2`. Further, the specific output, such as the estimated coefficients and their associated *p-values* for the proportional hazards models, can be obtained by the statement `statObj2.out.get(COEFFFICIENTS` and `statObj2.out.get(PVALUE)`, respectively, and then were assigned to the data members `coefficients` and `pValue`. The results obtained were quite consistent with the ones given in the books. Further, the expressions for fitting the other regression models, including logistic regression models, log-linear models, and P-spline regression models, are analogous.

```
import static java.lang.System.out;
import java.util.*;
import javastat.*;
import static javastat.util.Argument.*;
```

```
import static javastat.util.Output.*;
import static javastat.util.RegressionType.*;
public class RegressionModelsExample
{
  public static void main(String[] args)
  {
    double [] response = {58, 105, 88, 118, 117, 137, 157, 169, 149,
      202};
    double [][] covariate = {{2, 6, 8, 8, 12, 16, 20, 20, 22, 26}};
    double [] survivalTime = {156, 1040, 59, 421, 329, 769, 365, 770,
      1227, 268, 475, 1129, 464, 1206, 638, 563, 1106, 431, 855, 803,
      115, 744, 477, 448, 353, 377};
    double [] survivalCensor = {1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1,
      0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0};
    double [][] survivalCovariate =
      {{1, 1, 1, 2, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 1, 1, 1, 1, 1,
        2, 1, 1, 2, 2},
       {66, 38, 72, 53, 43, 59, 64, 57, 59, 74, 59, 53, 56, 44, 56, 55,
       44, 50, 43, 39, 74, 50, 64, 56, 63, 58}};
    Hashtable argument = new Hashtable();

    /* Fits a linear regression model */
    StatisticalAnalysis statObj1 = new RegressionModels(
      argument, response, covariate).statisticalAnalysis;
    /* Fits a Cox proportional hazards regression model */
    StatisticalAnalysis statObj2 = new RegressionModels(
      argument, survivalTime, survivalCensor,
      survivalCovariate).statisticalAnalysis;

    double [] coefficients = (double[]) statObj2.output.get(COEFFICIEN-
       TS);
```

```
    double [] pValue = (double[]) statObj2.output.get(PVALUE);
    out.println("\n" + statObj1.output.toString());
    out.println("Coefficients: " + coefficients[0] + " " + coefficients
       [1]);
    out.println("P values: " + pValue[0] + " " + pValue[1]);
  }
}
```

```
Output:
Linear regression:
{R_SQUARE=0.902733630006348, F_STATISTIC=74.24836601306407,...,
 F_PVALUE=2.5488662852901633E-5,...}

Cox proportional hazards regression:
Coefficients: -0.7959324065540736 0.1465698417095117
P values: 0.20856753917100646 0.0013912304425554023
```

In addition to statistical methods, a variety of plots can be displayed by only using the class `StatisticalPlots`, as illustrated by the following code. The data of interest are a sample of 50 computer purchases. The qualitative data and associated frequency distribution are assigned to the data members `category` and `data`, respectively. The generated pie plot can be assigned to the data member `graphicalAnalysis` and saved to the data member `plot`. The plot is put in a frame `pf` and then is displayed by the statement in the next line. For the other plots, the users can specify the required arguments and modify the statement for generating the pie plot, for example, `argument.put(PLOT_TYPE, BAR)` for creating the bar plot. A variety of plots generated by the class `StatisticalPlots` are presented in Figure 2.

```
import java.util.*;
import org.jfree.chart.*;
import statgraphics.*;
import static statgraphics.util.Argument.*;
```

```
import static statgraphics.util.PlotType.*;
import statgraphics.util.*;
public class StatisticalPlotsExample
{
  public static void main(String[] args)
  {
    /* Specifies the required argument and data */
    String[] category = {"Apple", "Compaq", "GateWay 2000",
                         "IBM", "Packard Bell"};
    double[] data = {13, 12, 5, 9, 11};
    Hashtable argument = new Hashtable();

    /* Specifies the required plot */
    argument.put(PLOT_TYPE, PIE);

    GraphicalAnalysis graphicalAnalysis = new StatisticalPlots(
      argument, category, data).graphicalAnalysis;
    JFreeChart plot = graphicalAnalysis.plot;
    PlotFrame pf = new PlotFrame("Bar Plot", plot, 500, 270);
    new PlotFrameFactory().putPlotFrame(pf);
  }
}
```

For using a variety of tests, regression methods, and statistical plots, only three classes `StatisticalTests`, `RegressionModels`, and `StatisticalPlots` are required. These three classes , playing the role of the class `Method` in the Data Analysis pattern and mainly handling the input arguments and data, are not very complicated. The examples for the use of the three classes can be found in the directory `examples`.


Figure 2 Plots Generated by StatGraphics

### 3.3.2 Regression model selection criterion for the data with correlated errors

A database management system (DBMS) is a software package for managing a computerized database. There are several advantages of using the DBMS approach (Elmasri and Navathe, 2007, pp.17–22). A relational DBMS implements the relational model, which was proposed by Codd (1970) and enjoyed great popularity (Elmasri and Navathe, 2007, p.141; p.233) due to its simplicity, mathematical foundation, and the structured query language (SQL).

In Wei (2009), a class of regression model selection criteria for the data with correlated errors was proposed. The algorithms for calculating a variety of model selection methods are quite similar, mainly depending on the weighted residual sum of squares and penalty function. Now, suppose a statistician wants to construct a class of robust weighted criteria (Hampel et al., 1986, p.367) in weighted linear regression or smoothing based on the robust weighted residual sum of squares, the simulated data and the real data were stored in two DBMS, MySQL (4.0.15) (`http://dev.mysql.com/downloads/`) and PostgreSQL (8.0.0) (`http://www.postgresql.org/`), respectively. In addition, the original data are available from `http://web.thu.edu.tw/wenwei/www/mysqlData.txt` and `http://web.thu.edu.tw/wenwei/www/postgreData.txt`. This statistician only needs to create the following class, which plays the role of the class `Method` in the Data Analysis pattern.

```
import static java.lang.System.*;
import java.util.*;
import static javastat.regression.PsiFunction.*;
import static javastat.regression.SelectionCriterion.*;
import javastat.regression.*;
import static javastat.util.Argument.*;
import javastat.util.*;
public class RobustSelectionCriterionExample
  extends SelectionCriterionTemplate{
  public double c;
  public double[] residuals;
```

```
public RobustSelectionCriterionExample(double c)
{
  this.c = c;
}

public Double weigthedRSS(Hashtable argument, Object ...dataObject){
  residuals = ((WeightedSelectionCriterion)
    new WeightedSelectionCriterion(argument, dataObject).
    statisticalAnalysis).residuals;
  weightedRSS = 0.0;
  /* Robust weighted residual sum of squares */
  for(int i = 0; i < residuals.length; i++)
    if(Math.abs(residuals[i]) <= c)
      weightedRSS += Math.pow(residuals[i], 2.0);
    else
      weightedRSS += 2.0 * c * Math.abs(residuals[i]) -
        Math.pow(c, 2.0);

  return weightedRSS;
}

public Double penalty(Hashtable argument, Object ...dataObject)
{
  return new WeightedSelectionCriterion().penalty(argument, dataObject);
}

public static void main(String[] args){
  DataManager dm = new DataManager();
  BasicStatistics bs = new BasicStatistics();
  Hashtable argument = new Hashtable();
```

```
  /* Gets the simulated data from MySQL */
  String[][] mysqlData = new DBLoader("jdbc:mysql://localhost/data",
    "root", "", "SELECT * FROM data").data;
  double[][] data = dm.transpose(dm.stringToDouble(mysqlData));
  double[] response = data[0];
  double[][] covariate = dm.getData(1, data.length - 1, 0,
    data[0].length - 1, data);
  double[][] weightMatrix = dm.inverse(bs.covarianceAR1(
    response.length, 0.2, 1));
  /* Gets the robust AIC in weighted regression */
  double robustAIC = (Double) new RobustSelectionCriterionExample(1.5).
    weightedSelectionCriterion(argument, weightMatrix, response, covari
    ate);
  out.println("The robust AIC is " + dm.roundDigits(robustAIC, 3.0));

  /* Gets the real data from PostgreSQL */
  String[][] postgreData = new DBLoader(
    "jdbc:postgresql://localhost/weitest", "wenwei", "1234",
    "SELECT * FROM data").data;
  data = dm.transpose(dm.stringToDouble(postgreData));
  response = data[0];
  double[] covariate2 = data[1];
  argument.put(SMOOTHING_PARAMETER, 1000);
  argument.put(DIVISIONS, 10);
  argument.put(DEGREE, 3);
  argument.put(ORDER, 2);
  /* Gets the robust GCV in spline smoothing with c=1.5 */
  double robustGCV = (Double) new RobustSelectionCriterionExample(1.5).
    weightedSelectionCriterion(argument, response, covariate2);
  out.println("The robust GCV is " + dm.roundDigits(robustGCV, 3.0));
}
```

```
}
```

```
Output:
The robust AIC is 1.329
The robust GCV is 1.126
```

The class `SelectionCriterionTemplate` in the package `javastat` defines the abstract methods *weightedRSS* and *penalty* and implements the default method *weightedSelectionCriterion*. This class plays the role of the class `StatisticalAlgorithm`. The data member `c` is the benchmark for the truncation of the residuals, while the associated residuals are assigned to the data member `residuals`. The robust weighted residual sum of squares can be obtained by loops and assigned to the data member `weightedRSS`. Since various steps for computing different selection criteria have been implemented by the class `SelectionCriterionTemplate`, the two required methods *weightedRSS* and *penalty* can be created by the statistician with only a few additions.

To obtain the robust selection criteria with the default method *main*, the class `DBLoader` can be used to loads the data from MySQL and PostgreSQL first and then these data are assigned to the data members `mysqlData` and `postgreData`, respectively. The loaded string data are converted to double arrays and then are assigned to the data member `data`. The responses for fitting the weighted linear regression model and penalized regression model are assigned to the data member `response`, while the covariates are assigned to the data members `covariate` and `covariate2`, respectively. The variance-covariance matrix for fitting the weighted linear regression model can be obtained and then be assigned to the data member `weightMatrix`. The robust AIC criterion for the weighted regression model can be computed by using the constructor `RobustSelectionCriterionExample` with the pre-specified truncation constant equal to 1.5. On the other hand, after using the data member `argument` to specify the the value of the smoothing parameter equal to 1000, the number of intervals on the x-domain equal to 10, the degree of the piecewise polynomial equal to 3, and the number of the differences equal to 2, the robust GCV criterion for the penalized regression model can be obtained similarly.

## 4. Design Patterns and R

The design patterns introduced in Section 2 can be deployed using R. In fact, the package `sandwich` introduced by Zeileis (2006) is an example of the modification of the Template Method pattern. The role of function `sandwich` in the package is similar to the method *TemplateMethod*, while the functions *bread.* and *meat.* in the argument list and the function *estfun* play the role of *PrimitiveOperation* in the class `AbstractClass`. Similarly, based on the S4 class, the code for the example in Section 2.1 can be rewritten in R, as illustrated in the following.

```
setClass(
  "AnalysisTemplate",
  representation(getStatistics = "function",
                 output = "function",
                 createPlot = "function",
                 doAnalysis = "function"),
  prototype = list(
    doAnalysis = function(response, covariate,
      getStatistics, output, createPlot){
      statistics =
        getStatistics(response, covariate)$statistics
      output(statistics)
      createPlot(statistics)
    })
)

setClass(
  "LinearRegressionAnalysis",
  contains = "AnalysisTemplate",
  prototype = list(
    getStatistics = function(response, covariate){
      list(statistics = lm(response ~ covariate))
```

```
    },
    output = function(...){
      /* Saves the result in a file */
              ...
    },
    createPlot = function(obj){
      /* Creates two plots */
              ...
    })
)

setClass(
  "LogisticRegressionAnalysis",
  contains = "AnalysisTemplate",
  prototype = list(
    getStatistics = function(response, covariate){
      list(statistics = glm(response ~ covariate,
           family = binomial(link = logit)))
    },
    output = function(obj){
      /* Displays the result on the terminal */
              ...
    },
    createPlot = function(obj){
      /* Creates a residual plot */
              ...
    })
)
```

The class `AnalysisTemplate` defines the required functions *getStatistics*, *output*, and *createPlot*. In addition, the function *doAnalysis* in the class implements var-

ious steps of the algorithm. The class `AnalysisTemplate` and its two subclasses `LinearRegressionAnalysis` and `LogisticRegressionAnalysis` play the same roles as their counterparts in Section 2.1. The key difference between the two languages in implementing the design pattern is that the functions playing the role of the method *PrimitiveOperation* need to be passed as the arguments into the function playing the role of the method *TemplateMethod*. The R code for the other patterns with the examples similar to the ones introduced in Section 2.2 and Section 2.3 along with the complete code for the Template Method pattern can be found in the directory `R-examples`.

## 5. Discussions

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems, as indicated by GoF (1995, p. 360). A good design pattern can be deployed successfully in a software project. In addition, refactoring based on some design patterns can be used to improve the quality of the original code. Based on the proven patterns, the proposed pattern aims to provide a sensible solution for data analysis. Although the proposed pattern is mainly applied to statistics, it can be also applied to numerical analysis involving input data, algorithms, and output in other branches of science.

Although there are several advantages of using design patterns, design patterns is not a panacea. A commonly used design pattern might be out of favor over time or in new context (Brown et al., 1998, pp.15–18). In such case, the solution provided by the design pattern might be problematic, i.e. "antipattern" solution. One possible solution is to use the antipatterns technique, which identifies the cause and provides the refactored solution.

`StatGraphics` is based on a high-quality open-source library `JFreeChart` (see Gilbert and Morgner), which consists of a variety of charts. More plots based on the package can be added to `StatGraphics`. `JavaStat` performs the probability calculations based on a high-quality package `JSci.maths.statistics` of an open-source library `JSci` (see Hale et al.). The primary goal of the library tries to achieve an accurate representation of the underlying science, including biology, chemistry, mathe-

matics, and physics. Thus, the statisticians could develop new methods incorporated with the ones used in other branches of science.

## References

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S. (1977). A Pattern Language. *New York: Oxford University Press*.

Anderson, D. R., Sweeney, D. J., and Williams, T. A. (2001). Contemporary Business Statistics with Microsoft Excel. *South-Western*.

Brown, W. J., Malveau, R. C., McCormick H. W., and Mowbray, T. J. (1998). Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis. *New York: John Wiley and Sons*.

Chambers, J. M. (1999). Computing with data: concepts and challenges. *The American Statistician*, **33**, 73–84.

Chambers, J. M. (2000). Users, programmers, and statistical software. *Journal of Computational and Graphical Statistics*, **9**, 404–422.

Codd, E. (1970). A relational model for large shared data banks. *Communications of the Association for Computing Machinery*, **13:6**.

Collett, D. (1994). Modelling Survival Data in Medical Research. *New York: Chapman and Hall*.

Elmasri, R. and Navathe, S. (2007). Fundamentals of Database Systems. *Addison Wesley*.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). Refactoring: Improving The Design of Existing Code. *Reading, MA: Addison-Wesley*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). Design Patterns: Elements of Reusable Objected-Oriented Software. *Reading, MA: Addison Wesley*.

Gilbert, D. and Morgner, T. *JFreeChart*, URL:`http://www.jfree.org/`.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). The Java Language Specification. *Addison Wesley*.

Hale, M., Cannings, R., Cross, D., Kooten, J. V., Lemire, D., Smith, T., Carr, J., Gonccalves, R. A., Dietrich, B, and Martin-Michiellot, S.. *JSci*, URL:`http://jsci.sourceforge.net/`.

Hampel, F. R., Ronchetti, E. M., Rousseeuw, P. J., and Stahel, W. A. (1986). Robust Statistics: The Approach Based on Influence Functions. *New York: Wiley*.

Krätzig, M. (2007). A software framework for data analysis. *Computational Statistics and Data Analysis*, **52**, 618–634.

Liang, F. Q. and Huang, C. W. (2009). Java micro edition technology for statistical and graphical analysis. *Journal of the Chinese Statistical Association*, **47**, 159–173.

McCullagh, P. and Nelder, J. A. (1989). Generalized Linear Models. *Chapman and Hall*.

Reese, W. (2000). Database Programming with JDBC and JAVA. *O'Reilly*.

Sommerville, I. (2007). Software Engineering. *New York: Addison-Wesley*.

Tate, B. (2002). Bitter Java. *Greenwich, CT: Manning*.

Warnes, G. R. (2002). HYDRA: A Java library for Markov chain Monte Carlo. *Journal of Statistical Software*, **4**, URL:`http://www.jstatsoft.org/v07/i04/`.

Wei, W. H. and Chen, G. J. (2008). JavaStatSoft: Design patterns and features. *Computational Statistics*, **23**, 235–251.

Wei, W. H. (2009). On regression model selection for the data with correlated errors. *Annals of the Institute of Statistical Mathematics*, **61**, 291–308.

West, R. W., Ogden, R. T., and Rossini, A. J. (1998). Statistical tools on the world wide web. *The American Statistician*, **52**, 257–262.

West, R. W., Wu, Y., and Heydt, D. (2004). An introduction to StatCrunch 3.0. *Journal of Statistical Software*, **9**, URL:http://www.jstatsoft.org/v09/i05/.

Zeileis, A. (2006). Object-oriented computation of sandwich estimators. *Journal of Statistical Software*, **16**, URL:http://www.jstatsoft.org/v16/i09/.

[ Received April 2010; accepted September 2010.]