# Interactive Education

Deborah Nolan
Department of Statistics, U.C. Berkeley
Duncan Temple Lang
Bell Labs Innovations

October 5, 1998

**Abstract**

Advances in computer technology are changing the way we teach. The explosion of Internet activities and the new GUI and WEB-oriented programming languages such as Java offer exciting opportunities for developing new instructional material. The project described here aims to utilize these new computing capabilities to provide a flexible and extensible environment for instructors to develop serious, portable teaching applications. In this paper we present examples of labs that we have developed, and discuss pedagogical considerations that led to their designs. We also describe a toolkit we have created for use in developing the labs, and we discuss how the toolkit supports lab development.

## 1 Introduction

Every year three to four thousand students enroll in statistics courses at U.C. Berkeley. The statistical training in these courses offers these students a rare opportunity to learn how to think critically (and statistically) about quantitative problems that are important to them. The aim of the project described here is to develop labs to accompany these courses that capitalize on the pedagogical advantages of new multimedia technology in order to convey some of the basic concepts of statistics in an interesting, useful and fun way.

The project includes four labs that represent diverse pedagogical techniques and a versatile toolkit that supports instructor development of labs through the use of simple scripting languages and properties files. We call it TILE – Tools for an Interactive Learning Environment.

In the course of developing the four labs, we have gained experience with pedagogical considerations surrounding the translation of teaching activities into an effective interactive educational computing environment. The labs attempt to extend the statistics syllabus by capitalizing on the pedagogical advantages of the computer, such as large data sets, computational speed, simulation, visualization, and animation. In the process, we have found one of the most difficult aspects of designing a lab is to determine how to combine these capabilities with multimedia technology. Issues pertaining to the implementation of statistical methods has been only one part of the lab development. We have also focused heavily on how to employ these methods in a topical context that emphasizes statistical concepts, and on how to design an interface that is helpful, intuitive, flexible and educational.

Our work in designing these labs has gone hand-in-hand with the development of the toolkit. The toolkit was born out of the motivation to provide a flexible and extensible environment to support the development of these teaching applications. New technology opened the door to an interactive and individualized learning environment, which in turn shaped the design of a toolkit to support instructor development of course material through simple scripting languages and properties files. The toolkit was also motivated by the need to see prototypes on the screen in order to evaluate them and to turn prototypes into production software that could be run on the primary different platforms and in Web browsers.

In the next section, we present a few of our considerations in the development of the labs that explore different statistical concepts and pedagogical techniques. After that we present two example labs, which serve as examples of the pedagogy, how an instructor customizes a lab, the toolkit design, and how to program a lab using the toolkit.

TILE has its origins in an innovative educational project originally undertaken by Professors Breiman and Purves of the Statistics Department at the University of California at Berkeley and partially funded by the National Science Foundation. Over the course of three years, we worked closely with Professors Breiman and Purves and several graduate students and staff of the Department on designs for 12 labs. Our discussions with them have had a significant impact on work presented here.

# 2 Pedagogical Considerations

Our goal in developing these labs is to teach students how to think statistically about quantitative problems that are real and important to them. We are not concerned as much with statistical methodology, or software expertise, as with concepts. We follow in spirit the approach of *Statistics*, by Freedman, Pisani, and Purves [1], but the labs can be used in conjunction with a variety of statistics courses and textbooks. They are stand-alone units that are meant to be used outside the classroom, not as a replacement for the instructor. Although we think that multimedia is useful to the curriculum, we also believe that value remains in having instructors teach courses using textbooks.

Our efforts have concentrated on the translation of teaching activities into an effective interactive, educational, computing environment. We have focused on how to employ topical contexts that naturally emphasize statistical concepts, and on how to design an interface that is helpful, intuitive, flexible, and educational. We have made the labs both context-based and context-adaptable so instructors can tailor the material to the interest of the students.

Each lab is substantial, and more involved than traditional questions concluding the chapter of a text. Students may complete them over several computer sessions. Accordingly, they are not calculator-like environments with which a student simply performs a statistical procedure. Rather than being an interactive component of a textbook, these labs extend the syllabus by putting techniques learned in the classroom in a more interactive and applied setting, where students have the opportunity to come up with their own solutions.

We have found animations are useful for representing a behind-the-scene calculation such as the simulation of 1,000 draws from a pool of tickets. Animation can also be effectively used to run a virtual experiment in order to see how data are created, and to connect a data point with a measurement taken on a subject.

Simulation is a powerful tool for demonstrating probability laws such as the central limit theorem and for finding the distribution of a test statistic. However, it can also be boring and nonintuitive. To address these concerns, we have included animations with simulations in order to help Describe the process, and we have limited the use of simulations as a computational device.

To maintain consistency across labs, we settled on a single interface that minimizes the complexity of the application. This interface uses a work and a help window and provides easy navigation between these and the elements of the work window. Additionally, quiz and animation controls are presented in a common way. The overall interface is similar to several Internet browsers with which many students are already familiar. The navigation system also allows students to easily move between exercises, and the help system supports dynamic hints and updates on status. The quiz system gives a student customized feedback, where quiz questions are immediately graded and students receive feedback that depends on the responses given.

The labs are designed to support a variety of levels of expertise. For example, the Design of Experiments lab described in the next section uses several practice rounds to introduce the ideas and a more difficult challenge round to encourage the student to think more deeply about the concepts. Another lab on observational studies is similar in spirit to an adventure game offering tasks at different levels of expertise as students gain familiarity with the dataset being anaylzed and the concept of confounding factors.

We provide more concrete examples of these pedagogical considerations in the examples below. We believe this approach has great potential for fostering students' interest in the material they are learning. Our aim is to promote statistical thinking and problem solving through interactive adventures. We also believe that this approach to developing an interactive learning environment is adaptable to fields other than statistics, and that it has potential for use at the high school level as well as the collegiate level.

# 3 Examples

In this section we describe two of the labs in TILE, one on hypothesis testing and one on experimental design. The hypothesis-testing lab is the simpler of the two. It serves as a case study for the process of programming a lab via the toolkit modules, which appears in the final section of this document. The second example, on experimental design, is more ambitious both in terms of design and programming. It serves as an example of how to build on the groundwork laid by the modules in the toolkit to develop a more complex lab.

## 3.1 Hypothesis Testing

The core of this lab centers on aiding the student in identifying and using the probability model in a hypothesis test. From a nontechnical description of an experiment and null hypothesis, the student is expected to be able to

- identify the chance process underlying the experiment,
- find an equivalent expression of this process in terms of drawing tickets from a pool,
- construct (via simulation) an approximate sampling distribution for the test statistic produced by the experiment,
- interpret p-values and significance levels.

At the start of the lab, after some brief instructions, the student is presented with a set of experiments in the Topics window (Figure 1). The topics window contains a table of contents, listing the experiments the student is expected to complete. In the example shown here the student is presented with two experiments. Each is accompanied by a short description of the problem. The instructor can choose any number of experiments for the lab, and they can be selected randomly for each student from a subset of those available for the lab. With random selection, students have less incentive to share results and they get a greater diversity of scenarios from which to learn the statistical abstractions. To change an experiment or to add a new experiment to the lab, the instructor makes the appropriate changes and additions to input files. Examples of these input files appear in the following sections.

The enclosing window interface in Figure 1 is used in all of our labs, in an effort to provide a familiar and consistent appearance to the student. This helps the student get started quickly on the content of the lab rather than wasting time trying to figure out the controls. Standard tools (quit-and-save-session, print, help, etc.) are available in the menu bar along with lab-specific actions. A message area and assistant, from which the student can obtain hints, are located at the bottom of the window. The icons below the menu bar comprise the navigation bar allowing the student to move between experiments in any order. Stacking the experiments allows us to use a single work window. The only other window displays the hypertext help system; it contains pages on the different topics in the lab and a dictionary of statistical terms. Finally, tooltips or messages appear over most components.

Consider the second experiment appearing on the page in Figure 1. The goal there is to determine whether one cloth manufacturer produces better fabric than another. One manufacturer has been used for several years and is taken as the standard. The other will be considered for a trial period, after which the decision whether to switch manufacturers needs to be made. Before proceeding with the lab, the student must first venture an opinion as to the strength of evidence needed for him to recommend switching manufacturers. This request is made to encourage the student to think about the problem before proceeding, to introduce the notion of setting a critical level in advance of experimentation, and to involve the student in the decision making process.

After the student makes his guess, he proceeds to an animation where he conducts the experiment (Figure 2). In this example, shipments of cloth leave the mill in trucks, and at the intersection the student chooses whether or not to send the shipment on to the jeans plant. When shipments arrive at the plant, they are examined and tallied as defective or not. After all shipments have been tallied, a test statistic is computed. The animation helps the student identify the source of randomness in the experimental process, and helps him understand how the data are created and compiled to form the test statistic. In other labs, such as the Stat Park, the student can and is encouraged to rerun the animation as often as he pleases to get a sense of the chance process in the amusement park game. Here, instead, he is restricted to one run of the experiment and the resulting test statistic.

After conducting the virtual experiment, the student proceeds to the Sampler, which is the centerpiece of the lab (Figure 3). Each experiment, regardless of its context, can be translated into a probability model using the Sampler. Here the student chooses ticket values and quantities for the pool, and determines the number of tickets to be drawn from the pool, whether draws are made with or without replacement, and the operation to perform on the tickets drawn, e.g. sum or average. The goal is to have the Sampler mimic the experiment. To do this the student uses the assumption that the new cloth manufacturer is no better than the standard and the chance process in the experiment. This lab is designed to follow the Stat Park lab mentioned above. In that lab the student first encounters the Sampler, and gains experinece in describing a chance process using the Sampler.

Once the Sampler is parameterized, the students runs an animation of the simulation that determines the null distribution of the test statistic. The animation helps the student see his test statistic as one of many possible outcomes from the experimental process, which in turn helps him in evaluating the significance of the observed statistic. The animation is performed in a cartoon-like way where an assistant (Figure 3) draws tickets from the pool, computes the statistic, and repeats the process over and over hundreds of times (there is a fast forward version of the animation).

The student then proceeds to the Post Sampler (Fig 4) where he is asked to consider the opinions offered by experts and to reconsider his original cut-off. He is encouraged to interpret the cut-off level that he and others provide in light of the null distribution. The experts' opinions are offered in a table (Figure 4). Each opinion is accompanied by either an $\alpha$-level or critical-level. The student fills in the associated missing critical or $\alpha$ level by reading tail probabilities from the histogram of the null distribution. To aid the student, tail areas are calculated interactively by moving a slider along the horizontal axis, and the tail areas are entered dynamically into the talbe by clicking on the appropriate cell.

The opinions offered by the experts include both traditional statistical statements concerning $p$-values and $\alpha$-levels and expressions we have heard students give as their reasons or gut-feelings for selecting a particular cut-off. Also at this time, we ask the student to draw a conclusion, choosing between a statistically sound decision and an adhoc remark. These different sources of opinions and conclusions are purposely juxtaposed in hopes that students will eventually be able to discern between them.

## 3.2 Designs of Experiments

We developed this lab in an attempt to convey to students some basic elements of an experiment, be they medical or industrial, etc. For the source of our narratives, we use excerpts of newspaper articles, accompanied by summaries of journal articles. In our experience working directly with news and journal articles in the classroom [2], we have found this medium to be effective in developing students quantitative literacy. It's also popular among students, because they can relate the stories to their own lives. By comparing and contrasting news and journal reports on an experiment, students hone their skills in critical thinking, synthesizing multiple sources and identifying important statistical elements.

From a nontechnical description of an experiment, we want the student to be able to answer the following questions:

Who were the subjects?
How were they split into treatment groups?
What was the treatment?
What response was measured on the subjects?

Additional important lessons we expect students to take away from this lab address the importance of having comparison groups; randomization; blindness; and generalizing results to populations other than those studied.

We try to present experiments on interesting topics with unusual features or that offer important lessons on the above topics. For example, many students have trouble identifying the subjects of an experiment concerned with the effect of equipping fishing nets with acoustic alarms on the number of porpoises that get entangled in the nets (see Figure 6). The students are more likely to think the porpoises are the subjects of this experiment than the nets. After all, its the porpoises that are getting caught in the nets!

To begin work on an experiment, the student selects an experiment from the topics window (Figure 5). This brings her to the main work area shown in Figure 6. In the practice rounds, the design is displayed on the right hand side in the form of a tree. The different levels of the tree represent the different stages in the experiment - enrollment, randomization, treatment and measurement. Where there are two boxes in a given level, the observational units have been separated by some earlier splitting criteria giving us different groups.

When the student first sees the design, the boxes are empty. The task is to fill in the boxes in the tree with the details of the experiment by relating the content of the newspaper and journal articles to the different stages. They do this by identifying text in the articles that relates to the particular box they are working on. For example, the phrase "number of porpoises caught" is represented by the porpoise icon in the Measure box, and "randomly" by a coin in the Split stage. When the student clicks on the phrase, it is entered into the box pictorially as an icon. All the phrases in the text relating to this icon are highlighted in the color of the box as if the student used a highliter to mark important passages in the text. If the student selects text that doesn't go into the box, a message giving some indication why the match is inappropriate is displayed in the message bar at the bottom.

After completing the contents of the design, the student moves to the animation component of the experiment. Here the design is displayed in a cartoon-like way and the student sees the subjects - shown as little green faced fellows in Figure 7 - participate in the experimental process. Some are rejected at the enrollment gate. Others move through the different stages being randomized in the centerfuge, treated in the blue box, and measured on the scale. Finally, the subjects fall into the appropriate histogram, adding their value to the appropriate bin. This animation helps to illustrate the whole process to the student and connect the data point to a particular subject. This allows the student to think of data as coming from a process rather than a table at the end of a textbook's chapter.

Once the virtual experiment is complete, the student proceeds to answer questions about the design. The purpose of these questions is to focus the student on some of the more important aspects of the experiment. For example, one question that follows an experiment about emergency medical treatment for stab and gunshot victims asks about randomization. As it happens, the treatment a wounded individual receives is determined according to whether the date is even or odd, not by randomization. Students sometimes confuse the deliberate alternation of treatment assignments with a chance process.

The questions in the wrap–up of the experiment are not intended to be graded. They are grouped into pages, and the student submits one page of questions at a time (Figure 8). Her answers are checked immediately; a profile of the student is built from her responses; and she receives individualized and detailed comments to help clarify important issues before proceeding to the next page of questions. In the stab-wound example, the student is asked in a subsequent page for reasons why the experimenter may have chosen a nonrandom assignment procedure.

As in the Hypothesis Test lab, the student may work on the practice experiments in any order, completing them sequentially or simultaneously. Once she has finished all the practice rounds, she can move to the challenge round, where the goal is to designs her own experiment (Figure 9). She is given a glossary of terms each identifying: a characteristic of the subject, a procedure for splitting the subjects into groups, a type of treatment, etc. She is also offered several designs for her experiment. Once the design is filled in, she runs the animation, which produces responses at random according to a simple model for biases and interactions that is determined by the design. Finally, after the virtual experiment is run, a news story that summarizes and critiques her design is dynamically generated.

# 4 Toolkit

We now turn our attention to implementing these examples and other labs in general. Producing a lab consists of 4 steps:

1. designing the pedagogical ideas,
2. designing the interface for the student,
3. programming both of these, and
4. creating the input files (e.g. experiments, images, etc.).

The previous section discussed the first of these two steps for the Hypothesis Testing and Design of Experiments labs. Most designers will agree that these two steps are complicated and that using simple prototypes in the design process would be beneficial. Also, from our earlier experience considering a dozen different lab designs, we were able to distill components common to several applications. Accordingly, to aid the third development step, we have created a library of these common tools and components. To avoid having the lab designer and instructors return to the program developer for lab-specific changes, we decided that an important feature of the components was to allow the instructors to configure and script elements of the lab. A further constraint was to develop the library and labs in a platform-neutral manner. In short, the primary intent of the third step was to

> *provide a flexible and extensible environment in which to develop serious "portable" teaching applications supporting several different levels of programming competence.*

Much of the portability issue was solved by using Java as the programming language. We have found it to be an extremely useful tool with which to develop these more involved graphical applications relatively quickly. The choice of Java also allows us to run the applications within browsers even though our focus is on stand-alone applications.

The toolkit is a collection of 16 Java packages[1] or modules and consists of almost 300 reusable classes developed by us and depending on other freely distributable components. The modules provide a relatively comprehensive suite of tools with which one can create a complete lab. The work window has been described earlier and acts as a container into which the lab-specific components are placed. Its services include tooltips for any of the components, navigation between the different work pages, an assistant providing hints for these pages and the basic menu items (such as Print, Quit, etc.). Also, the internal framework takes care of processing command line arguments; providing hooks to connect to different course management tools; saving and restoring a student's session; finding input files; etc. The help window is a hypertext system that displays HTML files supplied by the instructor, etc.

The other modules provide tools that can be used to build up the work areas of the lab. These include classes for different plot types, an SGML[2] rendering component and parser, a dynamic animation rendering facility, a scripting language for generating dynamic text, a quiz component that provides automated grading and feedback, statistical tools for sampling, random number generation, etc. Additionally, there are many lower level tools such as a generic mechanism for managing asynchronous object

[1] A Java package is a collection of classes which are grouped together to provide some higher level functionality.

[2] SGML stands for a Structured General Markup Language and is the generic framework in which HTML is a particular instance. The next generation of Web browsers will use XML[3] which is a subset of SGML.

creation for improved responsiveness in a lab.

Perhaps the most unique thing about the toolkit relates to the final clause in the italicized sentence above. An enormous effort has been made to allow the content of each lab to be specified at run time through simple configuration files. These typically involve pairing names that the application understands and values which specify the content. These are basically *properties* of the application. In the Design of Experiment lab, they control application-level properties such as the list of experiments to select from, and experiment-level properties such as the files containing the newspaper article and the layout of the design area for a specific experiment. The application level properties are processed in a hierarchical manner allowing the toolkit properties to be overridden by the instructor at both the course and lab level.

While the configuration allows the instructor to specify different fixed sets of inputs, often greater resolution is required. For these cases, we have developed a general way of creating simple scripting languages which allow instructors to control the application without programming in Java. We have used this mechanism to provide several such languages to control dynamic feedback and visual animation, for example. The languages use SGML as the syntax since instructors are familiar with HTML. Each language introduces new tag names and attributes for those tags which are used to control the application in some prescribed manner. For example, a quiz uses the `question` tag, and elements of an animation scene might be introduced by the `component` tag, which has attributes such as *image*, *bbox*, etc. These tags and attributes in the input document are made available to the Java application using a single method. This approach has proven to be very effective, as it has allowed us to develop a lab concurrently with one of us doing the lower-level Java development to provide the framework and the other controlling the content. Given the relative ease of creating new input languages based on the SGML/XML syntax and the tree traversal mechanisms in the toolkit, we believe that such scripting languages significantly help in developing, maintaining, and customizing individual labs.

In addition to the labs, we have developed small applications which are interactive interfaces to these languages and modules. These serve as debugging tools for instructors. They allow an instructor to visualize and interact with a succession of quiz pages or an animation scene, for example, and refine the inputs to obtain the desired effect.

# 5 Toolkit Modules

In this section we will give an overview of the toolkit and focus on some of the more interesting and unusual modules. The toolkit is organized into two sets of modules or packages. The first set provides provides the foundation for the second tier. This foundation includes methods for processing configuration files and their individual values, finding URLs (files and images), different GUI components (e.g. spinbox, tooltips), methods for reading datasets from URLs, extensions to basic Java classes, etc. These can be considered powerful utility classes which in turn rely on the core Java classes and numerous third-party Java packages.

The second collection of modules are described briefly in the following table. We will discuss a few of these in the sections below.

## *Selected* Module Descriptions

| | |
|---|---|
| *Plotting* | A hierarchy of class providing static and interactive plots for the common types used in introductory statistics (i.e. scatter-plots, time-series, histogram, column and bar charts). These classes provide support for external customization (through properties files) of almost all the elements of each plot (data,fonts, colors, labels, ticks, background images,etc.). Also, plots can be associated with a component or treated as images. |
| *Animation* | A scripting language for describing nodes in a static scene and how the dynamic components move between these nodes. These scenes are then rendered and animated in a GUI component by the interpreter in this module. Numerous parameters specifiable in the scene description file control the rendering and arbitrary Java classes can be easily added as components in the animation. |
| *DynamicTextSubstitution* | An XML (an extension to HTML) which allows instructors to generate and/or display text based on what the student has done up to a particular point. This allows for dynamic responses tailored to the student and removes the need for this to be programmed into the lab. This can be used for grading, providing feedback to the student, constructing hints, etc. This is akin to Dynamic HTML and the data within the application at run-time is made available for the instructor's use. |
| *Quiz* | A quiz browser component that grades and provides feedback to the student. A quiz is divided into pages and, within a page, self-describing questions. These are provided as arbitrary HTML documents (typically including a form containing the questions) with an extended tag set which identifies questions and the list of possible answers for each. This also allows the instructor to specify the feedback using dynamic replacement (see above.) |

*Stat* — Statistical functionality such as regression and random number generation (from common distributions and finite sets).

*Sampler* — This is a visual component which provides an interface to the Box-models in [1] used to illustrate probability models and sampling. This consists of

- an HTML area containing objects that can be inserted into the box by clicking on them,
- an animation area in which the physical sampling is performed and computations performed,
- controls for controlling the sampling (with or without replacement, the number of draws, etc.) and interchangeable histogram and summary table of the current contents of the box,
- a histogram plot in which the results are constructed.

*DataManager* — A highly flexible and transparent manager for constructing and caching objects *asynchronously*. This is used throughout the labs to improve performance by availing of idle time to create data to be used in the future.

*BasicApplet*
*GenericApp* — These modules provide the basic application structure including a help facility, configuration processing, dialogs, etc and the second provides the common graphical interface and its components (e.g. Toolbar, status line, hint button, etc.) Most labs developed with the toolkit will be based on the *GenericApp*. Using the common interface, a lab developer need only create the "cards" for each view in the lab. This greatly simplifies the development process especially since most of a card's functionality is inherited from the toolkit.

*SelectionLab* — This module extends the generic lab, providing facilities for randomly selecting elements from different groups of inputs, constructing a topics page describing each selected input and optionally requiring user input before the student moves to a particular card.

In addition to the modules, the TILE distribution contains several tools to help develop the input files for the different scripting languages. These help in preparing syntactically correct inputs as well as interactively editing them to provide the desired display. These provide some debugging facilities for these scripting languages described below.

We now elaborate on some of these modules, describing them in greater detail.

## 5.1 Plotting Classes

The toolkit provides different plot types such as time series, scatter plots, histograms, bar charts, etc. All plots can be optionally made interactive and support facilities for: identifying data elements by clicking and rubber-banding; stretching axes for zooming; etc. From an instructor's point of view, a useful feature of the plotting classes is the ability to specify attributes in input property files. The plots understand a common set of attributes such as background and plotting colors, background image, axes ranges, number of tick marks, legends, fonts for the different labels, etc. Even the data can be specified as a URL or inline. Also, each class understands attributes that have meaning just for it, such as number of bins or bin widths for a histogram.

The inheritance of properties allows all plots within a series of labs to have a common appearance while individual labs or plots can be customized. Typically, the application provides the data to the plot using the common methods in the plot classes and the instructor controls the plot appearance. Default attributes are computed for each plot including the appropriate number of ticks, etc.

From a programming point of view, the plots are interesting because they can be used anywhere (e.g. in a button) since they are self describing objects. This approach differs from having a GUI component for one or more plots. The classes support the Java listener event model and can notify other objects when the plot has changed or the user has interacted with it in different ways. Also data can be added incrementally – a feature used in animations to show histograms and scatter plots being constructed dynamically. Finally, the plot classes are designed to be extensible due to the rich class hierarchy making it easy to create new types that inherit all of the features mentioned in here.

Each plot is (optionally) interactive. Axes are "stretchable" by dragging a tick so as to provide a zooming effect. Rectangular areas can be etched out to identify different data elements in a plot. Interactive histograms allow the number of bins to be altered. The classes support methods for the programmer to add, move, delete and identify data elements.

## 5.2 Animation

We use the animation facility in a variety of places throughout the labs. For example, the sampler module illustrates the process of drawing marked tickets from a pool with or without replacement to generate a sample and a statistic. In the Design of Experiments lab, the experiment is animated to generate the data for the subjects.

Such interactive dynamic displays are complicated and time consuming to program. Accordingly, we provide a simple abstraction of the animation scene by categorizing the elements into 3 groups: background/global attributes, static components, and elements that move between the static components. This has been sufficient to characterize all of the animations we have considered so far. Each element in the scene is identified by a particular SGML tag defining to which of the 3 categories it belongs. Attributes in the tag define the characteristics of the element such as its position and size, or a sequence of images to be displayed when it is animated, etc. The coordinates of the components are interpreted relative to the instructors scale specified in the `animation` tag. When the scene is rendered, the coordinates are scaled relative to the available space on the screen. The instructor can specify whether a component's bounding box is also scaled or fixed.

Most of the static components are simply images but there is also a mechanism to use an arbitrary Java object. We use this, for example, to allow the student to select a ticket from the sampler pool and to allow histograms to be placed as nodes in the scene (Figure 7). The attributes are passed to these objects which allows us to configure the histogram from the animation input file. This facility allows the animation to be used in a variety of situations as a primitive programming language.

The SGML syntax can also be used to define relationships between components and is used to describe design trees in the Design of Experiments lab.

The elements in the input file allow the scene to be rendered statically. The final part of the animation is the part that controls the dynamic action. This is programmed in Java where each of the moving components runs in its own thread. When each component arrives at an end point of its sub-animation it sends a message to the centralized manager which then decides globally what will happen next. This allows developers to extend the animation control relatively easily since the objects work independently with a single point of synchronization. Different variants of the basic manager class control along which paths the dynamic components move. In the Design of Experiments animation, the subjects are dynamic components and the paths they travel relate to the treatment which in turn is specified in input files as a stream of data for the 2 histograms. Finally, the animation controller randomly selects values using the random generator module and associates them with different subjects. When the subject reaches the end of the design, its value is added to the histogram. The animation classes provide the student with controls to start, stop, pause, fast forward and restart the dynamic animation.

The content of the animation input files is formatted using HTML-like tags. An instance of an animation scene is show in Figure 7. Sections of an example input file are given below. There are currently four active tags recognized by the animation scene parser. These are `animation`, `component`, `class` and `path`. Characteristics of each element can be specified as attributes to the tag or as properties within the tag (or a mixture of both).

The description starts with the global and background information via the `animation` tag. Information about the background image and color and default values for attributes to be used for the different elements are specified here. The most important attribute however is *size*. This is the resolution which defines the coordinate system in which the scene was designed. The location of the different elements are relative to these dimensions. This allows the instructor to define the units in a manner convenient to them. The animation interpreter determines the available size of the component in which the scene is being rendered and scales the distances appropriately.

The next part of the input file lists the different static components. Each component has a name so that it can be identified in a path specification described below. Note that there are two tag types denoting a static component. These are `component` and `class`. The first is the common type that defines a node described by an image. When the scene is rendered, the image is drawn at the node's position. During the animation, when a dynamic component enters the node, the sequence of images specified by the the *images* attribute are displayed. The interval between frames is taken from the *delay* property. This gives behavior much like an animated GIF image (GIF89). A short-hand syntax for image sequences is provided which is similar to the UNIX tcsh shell.

In some animations, the nodes must be more intelligent (e.g. supporting user interaction to determine which path to take from the node) and are more complicated than an image. We implement these as Java objects and specify them using the `class` tag and *class* attribute identifying the Java class. The animation interpreter creates an object of this class and initializes it using the node's attributes from the input file. Objects used in this way must implement a relatively simple Java interface which allows them to be moved and resized, to be queried for their in and out points, and to have parameters set from a *Properties* table object. Support for these classes makes the animation module a powerful non-programming tool for developing labs.

The location of a component is specified via a bounding box (the *bbox* property) which specifies the position, and optionally the width and height of the component. If the dimensions are not specified here, the size of the image determines the visual area of the component. If they are specified, the images are scaled to fit in this frame. Fixed components also have in and out points. These are used to give precise control over the position the dynamic components enter and exit the static node. These can be specified as coordinates (relative to the box or the scene) or as compass directions (i.e. N,NE,S,W,...). Finally, the *show* property is a boolean indicating whether the node should be displayed or not.

During the animation, a moving object is created and moves between nodes along paths. The path is selected from the available paths enumerated between the source node and all other static nodes. These are the final elements in the input file introduced by a `path` tag. The *components* attribute identifies the two components using their *name* attributes. The *show* attribute here determines whether the path's curve is drawn in the scene rendering, effectively making it visible or invisible. Similarly, the *color* determines the appearance of the curve if it is drawn. The *type* attribute determines the shape of the path. Several are available in the toolkit including Linear, Bezier and Point. Others can be added by extending the appropriate class path and initializing it from an additional attribute.

The remainder of the attributes in the `path` tag control the movement of the object along the path. The *images* attribute specifies a sequence of image frames which are cycled through along each step. The *delay* controls the interval between steps and the *steps* attribute controls the number of steps (and hence the size of a step) into which the path is divided.

Currently the *path*s are pairwise connections between nodes. In a future release, we will support more specific paths, connecting more than two boxes. This will allow an object moving through a scene to have different appearances controlled by its overall path rather than its current pairwise path. (This is currently possible only at the programming level.)

```
<animation background=QCScene size="890,453" simultaenous=5
           steps=40 sampleSize=10 relativeHotspots=true>

<component name="Start" bbox="1,1,0,0" show="false" >
    </component>
<component name="Enroll" images="door6,door[1-5],door[7-10]"
             bbox="160,25" inHotspot="W" outHotspot="E">
    </component>
<class class="Charts.InteractiveHistogramTool">
<!--  Attributes specified as properties. -->
name:Histogram
bbox:20,175,250,250
show:true
inHotspot: N
data: fileName
backgroundColor: blue
axisColor: green
XAxisLabel: Measurement
YAxisLabel: %
attributes: file
</class>

<path components="Start,Enroll" images="SmallGuy[1-9]"
```

```
       show="true" delay="10" color="green"></path>

<path components="Enroll,Split" images="SmallGuy[1-9]"
      show="true" delay="10" color="blue" steps=10></path>

</animation>
```

Note that the order in which the components and paths are specified is not important as the parser unravels the elements. Also, each element in the description supports the *depth* property. These are integers with larger values indicating greater visibility. Finally, the format of the file allows multiple scenes to be supplied in one file and even nested animation scenes. The latter allows sub-animations to be specified as components.

Each animation is controlled with a set of buttons. (These of course can be hidden via a property in the animation scene file.) The buttons provide the user with ways to start, stop, pause and fast–forward the animation.

Each of the dynamic components in the animation has its own *thread* of control. Messages are sent to a central manager when they reach the end of the path and when the static components have completed their sub-animations. This simplifies the overall synchronization and makes it relatively simple to modify the animation behavior.

## 5.3 Dynamic Text Substitution

As mentioned earlier, a major goal in designing the labs is to provide personalized questions and feedback to the student based on what he has done up to that point. While the instructor can change which file to display before the student runs the lab, elements of the document cannot be changed to include information that is only known when it is displayed. An obvious way to do this is to program the dynamic construction of the text in Java since this is where the variables are stored. This however is inconvenient and makes the application relatively rigid as the instructor can't easily change the phrasing and formatting of the text without assistance from the developer.

Our approach is to use an SGML tag through which the instructor identifies a location in the document at which text will be conditionally inserted at different times in the application. This `replace` tag supports different attributes that indicate the conditions under which the text should be replaced and the actual content of the additional text. Each of these attributes can refer to variables in the Java application that are easily made available by the developer as needed by the designer. The conditions and substitution are defined in a simple language that supports most of the usual comparison and logical operations. A few other attributes of the `replace` tag and features of the comparison language make this a very simple but powerful scripting language and allows the instructor to specify control flow of parts of the lab without using Java. The developer of a lab need only provide access to the variables by putting them in different tables and arranging to process the document at the appropriate times.

This mechanism is used in several labs. Quiz grading can provide feedback to the student based on his responses. In the challenge round of the Design of Experiments lab, the newspaper article is created by determining the type of design selected and the icons present in the design and then constructing sentences accordingly. A novel use of the mechanism in another lab allows us to transfer information from a plot to a table in a flexible and extensible manner with a minimal amount of programming.

The `replace` tag supports several attributes most of which are in the form of expressions in a simple but flexible scripting language created for the toolkit. The expressions refer to variables made available by the application contained in collection of tables provided when the replacement is performed. The expressions operate on these variables and their values by comparing values using the usual comparator operators ($<$, $>$, $>=$, $==$, $!=$, etc.) and functions provided by the table classes (e.g. exists() to ascertain if a variable is present). Values can be concatenated and a variety of different mathematical and increment operations performed. Additional functions can be added by extending the class (*Context*) that implements these tables. Discrimination between non-unique variable names is obtained by qualifying the variable name with the name of the table in which it is contained For example

```
page1@q2
```

identifies the variable `q2` in the table named `page1`.

The expressions return values based on the current values in the table. How these results are used depends on the attributes of the `replace` tag. There are two primary forms. If the *condition* attribute is provided in the tag, it is evaluated. If it returns *True*, the attribute *true* is evaluated and its result used as the text in the insertion. Alternatively, the *false* attribute is evaluated and inserted.

A non-binary form of the *condition* attribute is provided by *switch*. When it is specified, its expression is evaluated and the result identifies the name of another attribute. The evaluator searches the attributes of the `replace` tag for this name and evaluates its expression. If no such attribute exists, the *default* attribute is used or no text is inserted.

The simplest form of replacement involves no conditional insertion but simply maps the values in the table into a string. This is done via the *value* attribute. The expression is evaluated and the result converted to a string and inserted into the text. This might be used for example to display the page or question number in a quiz. Some expressions need not produce text to be inserted but are used for their side-effects. These are typically assignment expressions and can be specified using the *condition* attribute with no *true* or *false* attributes.

In some cases, the condition in `replace` tag is a composition of logicals involving the existence of a particular variable. For example, we might want to test if a page in a quiz has been graded and then determine if the question was answered correctly. The expression

```
exits(graded) && @q2_correct == T
```

does this, but will return false if the page is not graded. Then the value of the *false* expression will be inserted. If instead, we want to ignore this `replace` element unless the `exists(verb)` is true. This can be done using the attribute *when* and its value is an expression resulting in *True* or *False*. In this case, the replacement act is conditional rather than the content.

Finally, the *init* attribute identifies the expression that should be evaluated before actual substitutions are inserted into the document. This is mainly used to initialize variables using the assignment operator (`:=`) before the replacement is started. Also, some initialization should be done just once (such as the number of passes being created and set to 0). This can be specified with the value **once** for the `init` attribute.

The instructor and code developer must agree on the variables that are to be made available via the *Context* objects. These normally become apparent during the design stages of the lab. However, as inputs change new variables are required. These are very simple to add to existing context tables and new contexts can be added to the collection made available to the replacement evaluator. The values are only required when the replacement is performed so the developer can update the values as the internal variables are modified before performing the replacement. If certain complicated conditions or operations are being performed frequently in the input, Java methods can be provided in the contexts which perform these directly. The instructors can invoke these directly with an arbitrary number of arguments and the replacement evaluator will attempt to invoke that method. The argument to these methods is itself a *Context* object containing a named list of values. `<replace value="foo(@x, @y)">`

The expressions are parsed using lex and yacc equivalents in Java. A grammar for that language is part of the toolkit and can be modified or extended. The evaluation of the expression trees is performed by a handful of classes. All of this is separate from the HTML document which is parsed using the *HTMLparser*. The *DTSTree* traverses this parse tree and identifies the `replace` elements. These are then processed each time the replacement is performed. The *Context* class is simply a hashtable to which new methods can be added.

## 5.4 Quiz

Quizzes are used at different points in a lab to help the student distill the important lessons from that stage. While the responses can be readily used for grading purposes, our emphasis is more on providing dynamic feedback tailored to the students inputs to explain why certain responses are incorrect. In other words, it is very much part of the learning experience rather than an end–point.

The *Quiz* module provides an input format for the instructor, a simple visual interface for the student, and a grading and correction mechanism to provide feedback for the student. We describe each of these in that order.

Each quiz is divided into one or more pages and typically specified as a comma-separated list in a properties file. Each page is supplied as an HTML document providing the usual formatting control and contains one or more questions. A page typically contains a single HTML form displaying all of the questions. Questions are contained in a `question` tag with a *name* attribute. This identifies them to the program for grading purposes, etc. Checkboxes, radio-buttons and choice-menus are the most commonly components used to provide multiple choice questions. Text areas can also be used but require more care in grading. The forms work in the standard HTML manner associating a value with a name for a given component. The single form allows the user to complete all the questions on a page and then submit all of them in one action. The correct answer(s) for each question are provided in the document via a *hidden* input element using the same name as the question with the suffix `-ans` appended. This arrangement now identifies the user's and the correct answer to the quiz grading mechanism.

A simple example of this format is shown below.

```
<question name=Q1>
<font color=yellow> Randomization</font>:
The surgeons were randomly assigned to treatment groups.
<br><input type=radio name=Q1 value="yes"> Yes
<br><input type=radio name=Q1 value="no"> No
<input type=hidden name=Q1-ans value="yes">
</question>
<br>
<p>
<question name=Q2>
1. Each surgeon took the math test twice--once while listening to
music and once without music--that is why it is called a cross-over
design.  What do you think might be an advantage to using a
cross-over design?  Choose all that apply.
<p>
```

```
<dl>
<dd><input type=checkbox name=Q2 value="unlucky">
Without it you might get unlucky and have a
lot of surgeons with high blood pressure in the music group.

<dd><input type=checkbox name=Q2 value="twice">
It means you get twice as much data with the same number of
 surgeons.  It's like having twice as many subjects in the experiment.

<dd><input type=checkbox name=Q2 value="base">
Each surgeon's blood pressure while listening to music is
compared to his own blood pressure while not listening to music.
That should make it easier to detect any effect that music might
have on blood pressure.
<input type=hidden name=Q2-ans required="base" match=2 value="unlucky,base">
</dl>
```

This example also illustrates how different criteria can be specified to determine if a student's answer is correct. The *match* attribute indicates the minimum number of elements the answer must contain. The comma-separated list for the *value* gives the possible answers and the *required* attribute lists answers which are essential to be correct. Other values that are supplied are either optional or forced to match the list of possible values via the *strict* attribute. Other criteria are implemented in the grading mechanism and described in the documentation for this module.

Giving the quiz pages a common appearance is useful for the student. While a template document helps in this area, if an instructor later decides to change this template, she must change all the files. To circumvent this, HTML text to be inserted at the top and bottom of each page can be provided in the properties file. The same can be done for each question. These wrappers can contain information such as the page and question number, the total number of pages or questions, etc. This is a convenience feature and can be generalized using the `replace` tag of the *Dynamic Text Substitution* module to produce quite elaborate displays. The replacement mechanism can also be used to provide adaptive quiz content that builds on the students answers to earlier questions. Also, it allows later questions to be phrased in terms of the answers to earlier questions without the student being able to see this.

The visual interface is an HTML display and toolbar containing three buttons. The buttons provide navigation facilities for the pages of the quiz allowing the student to go backwards and forwards and also to submit the current page of the quiz. The student cannot view the next page in the quiz until they have submitted the current page. They can of course look at previous pages to check earlier answers and return to the active page. If they have not answered all the questions on the page when they submit, a message is displayed and they are prompted to complete them. When a completed page is submitted, it is then graded and feedback provided.

The default feedback is to display the total number of questions answered correctly at the top of the page and to mark each question as right or wrong. This is done by inserting an appropriate image beside the question title. A more elaborate response mechanism can be provided using the *Dynamic Text Substitution* to provide responses based on the individual and collective answers. For example, if the student selects the second box in Q2 of the example above, we might display some text mentioning the lack of independence of the measurements. The flexibility of the replacement mechanism allows elaborate responses to programmed by the instructor.

As with all elements of the toolkit, the interface can be readily extended to create a different appearance and functionality.

## 5.5 General Application Interface

As we explained above, our generic application consists of an introductory window, work window and a help window. This structure is provided by the *Generic Application* module. This offers a simple way to create an application and also takes care of many configuration issues and provides an easy mechanism for adding new properties understood by the application. Also construction of the different components of the work window can be performed asynchronously by passing the initialized card objects to the *Data Manager* class. When an application is running, different services are available via different methods in the base class. These services include displaying a help page, saving the state of the application, printing the contents of the active card, displaying messages and switching between cards. Menu items are available for most of these and are controlled transparently by the toolkit's classes. However, the methods can be invoked directly.

The configuration of the lab and toolkit works for a sequence of input files using the properties format (`name: value` pairs). There is a file for each of

- the TILE installation,
- the course (i.e. a particular collection of labs),
- the specific lab, and
- the student.

These are merged together to construct one set of properties, with those lower in the list overriding the earlier ones. The attributes configured by these properties include search paths for input files (including images, etc.), defaults fonts and colors, images, labels and tooltips for the different toolkit buttons, etc.

The initial screen is an extension of the help window and is used to provide an overview of the lab's goals and contents. Links in the documentation are handled as they are in the Help window. Similarly, images referenced in the document are found by searching the different directories in the toolkit's search path. The toolbar also contains a button to bring the work window to the foreground and allows the student to start the lab.

The help window contains a menubar, toolbar and HTML area. The menubar and toolbar contain mechanisms to jump between the different help topics which include the initial page, a glossary, help on help. Within each of these pages there are links to the other help pages allowing for an arbitrary structure to the documentation. There is also history support for browsing, which allows navigation between previously viewed pages

Now we turn our attention to the main window - the work area - shown in Figure 6. The menubar contains the usual action entry (for quitting & printing), a help entry providing access to the and different help pages in the help window. The navigation bar contains a button for each of the cards in the main window. While the card is being constructed, the cursor in the button appears as a clock rather than having the button greyed–out. Tooltip messages appear when the user places the cursor in the button and doesn't do anything for a short period. This is true even if the card is not ready for showing. The content of the message is read from the input files for each card. Also, any component can be registered with the application's tooltip manager provided by the toolkit. Both fixed and run-time/dynamic messages are supported.

The message bar is used as a place to display messages to the student including hints, errors and suggestions. This can use the principle of stacking the most recent messages being displayed for a period of time and removed to reveal earlier messages. Controls to operate on the message list are available. Messages are annotated by a sound and optionally a changing color to get the user's attention. Finally, the workman in the lower right corner can be used by the student to get immediate help on the current situation. Clicking on the workman generates a request to the active application card for a hint. This can provide different messages based on what the user has completed so far and how many times a hint has been requested. This allows the lab designer to help students who are having a difficult time in one way, while providing other messages to more advanced students.

From a programming point of view, creating a new lab involves describing the cards in the main window. This involves extending the class *GenericAppCard* (or one of its sub–classes). This class and the toolkit take care of constructing the entries in the menubar and toolbar for the card. How to construct the components in this card is described in the next section.

# 6 Programming a lab

In this section, we describe the development cycle of the Hypothesis Testing lab. Throughout this description we attempt to describe how one uses the toolkit and illustrate the different levels at which one can interact with the toolkit. It should become clear that the process of programming a lab will involve gluing components from the toolkit and that the harder aspects are the design of the lesson and interface and the creation of the inputs.

We are using this lab as our example material because it is reasonably simple to describe and illustrates many different aspects of the toolkit and pedagogical approach. We feel it is typical of the types of lab one might consider for an introductory statistics. It may be considered intermediately complicated since it is more involved than putting a single component on a Web page, but much less comprehensive than some of the other labs we have considered (e.g. Design of Experiments).

## 6.1 Programming

We must first create the class that will start the Java applets. We do this by extending one of the toolkit applet classes - *BasicApplet*, *LabApplet* or *ExperimentSelectionLab*. Our design fits the *ExperimentSelectionLab* class perfectly. (Hardly surprising since the toolkit was developed with some of these labs in mind!) The global configuration file specifies different groups of scenarios to select from and the number within each group. For example, the lines

```
numPracticeRounds: 1,1
experimentsA: EPA, ESP
experimentsB: Jeans, FDA
```

indicate that one experiment should be selected from the set EPA and ESP and one from Jeans and FDA. The applet selects from these randomly when it starts.

The *ExperimentSelectionLab* class also takes care of creating a card for each scenario and putting an entry in the navigation bar and Window menu. It does this by invoking the method *createExperiment()* in *ExperimentSelectionLab*. We override this method to create our own card for each scenario. This method is simply

```
public SelectionLab.ExperimentCardInterface createExperiment(
             String name, boolean challenge)
 {
  ExperimentCard e = new ExperimentCard(name, challenge);
 return((ExperimentCardInterface)e);
}
```

This simply creates a new card which we will define for this lab, extending the toolkits class *GenericExperimentCard*.

The developer has the responsibility of programming the details of the different cards. The information defining each scenario is provided by lab developer (or instructor) comes in the form of a properties table. This, again, is simply an ASCII file contain `name: value` pairs. One is shown below. The Java object representing the scenario is responsible for interpreting these properties or ignoring them. Some are understood by the toolkit such as the tool-tip and icon, which refer to the button in the navigation bar. We will discuss the others when we focus on the details of the card.

Note that in creating the card, only the properties table describing this scenario is constructed initially. The applet then takes care of managing the creation of the card's visual components asynchronously via the *DataManager* module and the card's *make()* method. This allows the lab to display the initial screen describing the lab and the topics page describing the experiments.

The *ExperimentSelectionLab* object constructs the topics page by querying the newly created scenario/card for its selection icon and description text. These are obtained from the scenario's properties file, and each property either names a file containing HTML text or contains the text itself. The applet concatenates these descriptions and images to create the topics page from which the user can select the different scenarios.

In this lab, each scenario has an extra description which consists of an HTML form. The user must make an initial statement about the critical value or p-value about which they would reject the null hypothesis. If the user attempts to move to a particular scenario before entering his guess, he is politely reminded of the need to enter a guess. This guess is used later in the lab.

Each of the entries in the property file for an experiment is interpreted by different parts of the application

```
\label{EPAProperties}
icon:   Images/exhaust
selectionIcon: Images/exhaust
tooltip: Car Emissions
description: Data/EPA/description.html
subDescription: Data/EPA/subDescription.html
minimission:
preStory:  Data/EPA/AnimationStory.html
ticketFile: EPA/Tickets.html
animation: Data/EPA/animation.html
histogram: Plot.properties
opinions: EPA/opinion.html
conclusion: EPA/conclusion.html
```

At this point, the introduction screen has been displayed and the user can click on the Go to Lab button to bring the main window to the foreground. This shows the description of each of the scenarios, formatted with HTML. The only development at this point is extending the *ExperimentSelectionLab* to create the lab instance. The inputs in the global and scenarios' properties files have provided all of the content. This is typical for the toolkit and the developer need only concern herself with the individual cards and their relationships.

## 6.2 The *ExperimentCard* Class

The *createExperiment()* method introduces the class *ExperimentCard* which we use to implement each scenario in this lab. This extends the toolkit's *GenericExperimentCard* which handles creating the properties table from the file and interpreting many of the contained properties. For example, it will create the text for the topics page from the **description**, **subDescription** and **selectionIcon** properties, the icon for the navigation, and so on ....

The primary constructor (which initializes an object of this class) inherits its actions from the the *GenericExperimentCard* which creates the properties table. The construction of the visual components are implemented in the *make()* method. In the spirit of encapsulation and good programming, this is a simple method. It extends the actions of the base class and simply creates a tab component and its three entries.

```
public boolean make() {
 super.make(); // call the base class' version of this method.

  /* Create one component for the entire work area. */
 add("Center", tabArea = new TabComponent());

  /* Create the different entries for the tab
     component. */

 pre =  new PreSampler(specs);
 sampler = new SamplerPanel(specs);
 post =  new PostSampler(specs);

  /* Add each of the components.  */
 tabArea.add("Pre-Sampler", pre);
 tabArea.add("Sampler", sampler);
 tabArea.add("Post-Sampler", post);

  /* Declare the construction is finished. When the
```

```
        student clicks on the navigation icon (or menu
        item) for this window, the application
        framework checks to see if the card
        construction has completed.  It also checks if
        a guess is required and if so, has one been
        submitted.  */

    completed(true);

     // announce that the card was constructed properly.
    return(true);
  }
```

Ignoring the details of how each of the 3 components is created for the moment, this completes most of the *ExperimentCard* class. (We will later use it trivially as a bridge to connect the different components.) We have done the minimum of programming that allows us to provide our own interface within a card. The toolkit and input files have taken care of all other details via extension. We now turn to the different components within the tabbed area and see how we glue components of the toolkit rather than inheriting from them.

## 6.3 Pre–Sampler

This component contains just two elements - an HTML widget displaying a description of the process and a cartoon-like display of the process. The first of these is very simple since the text needs only to be displayed (and default link handling performed.) This can be done by creating an *HTMLcomponent* object available from the toolkit. We pass the *File* containing the HTML text to the constructor and leave the toolkit to find it for us (so that the instructor doesn't have to hard code the location in the input files.) The following code creates the widget

```
 String fileName =
    GenericGlobals.findFile(specs.get("preStory")));
 new HTMLcomponent(new File(fileName);
```

where the object `specs` is the scenario's properties table passed to the constructor from *ExperimentCard*.

The second component is intended to allow the student to explore scenario's measurement process. Some of the scenarios we use simply have different subjects being selected for different treatments, measured and added to a histogram. Other scenarios, however, are more complicated in that the user can control the paths of the different subjects (up to a fixed number) entering each treatment. This would be prohibitive to program for a specific lab. However, the toolkit provides the *Animation* package to do just this. The toolkit provides classes that can be used as nodes in an animation and that allow the user to control the direction of subjects exiting the node. And the *Animation* allows the instructor to specify and modify the display (or scene) as an input file using a relatively simply scripting language. The following code creates this component and arrange for it to be displayed with controls for the user (i.e. Play, Pause, ...)

```
String tmp = specs.get("animation");
AnimationSceneDescription d =
      new AnimationSceneDescription(new File(tmp));
animation = new AnimationPanel(d);
```

If the control of the animation does not meet the needs of the lab (e.g. the available paths from a node), methods in the *AnimationManager* class can be extended and a class specified for the manager property in the animation input file.

## 6.4 Sampler

This is an entirely trivial component to implement since it is provided in the toolkit. We need only transfer the name of the file containing the ticket inputs from the scenario's properties table to the sampler. We extend the class *BasicSampler* provided in the toolkit and provide a constructor that extracts the **ticketFile** from the properties table specified by the **sampler** property in the scenario's *ExperimentSpecs* properties table.

We will also need to extract information from the sampler (namely the histogram) to use in the post-sampler component when the student completes the first two steps of the scenario. This information is available from a method (*()*) in the *BasicSampler* that we are extending. There is no additional programming needed.

## 6.5 Post-Sampler

Of the three components, this involves the most code. However, most of it serves to connect components of the toolkit together. There are three components, 2 of which are HTML displays and the last a plot. The first of the HTML components displays a table presenting some opinions about the scenario and columns for $p$– and critical–values corresponding to each opinion. The user is to complete the table by interacting with the plot, which is an interactive histogram containing the data from the sampler panel. The user can click on it to identify a particular value on the x-axis. Depending on which column the active cell in the table is, the critical– or P-value corresponding to this value is inserted.

The final component consists of text and an HTML form asking the student whether the hypothesis should be rejected. This can be submitted only after the table has been completed.

We have seen how to create an HTML widget. For the conclusions panel, we can use this widget but we need to catch the form data when it is submitted. There are two basic approaches for doing this. The first is to extend the *HTMLcomponent* class and override the method *handleForm()*. The regular method returns a *Hashtable* containing the submitted values which can the be processed in the grading of the scenario. The second approach is to arrange to have the *PostSampler* object notified when the form is submitted. We do this using Java's listener paradigm. This avoids creating a new class, but we still need to provide a *handleForm()* method and invoke the *addFormListener()* method for the component.

The first HTML component appears to be a little more challenging. We want to allow the instructor to provide an arbitrary HTML document so that she can change its appearance without modifying any code. However, we must identify the active cell and also insert the appropriate value from the plot when it is selected. We achieve the first requirement with HTML references -

```
<a href=julia>text</a>
```

We extend the class *HTMLcomponent* and override the method *handleLink()*. This method simply stores the name of the link as the active cell. (If a value has been selected in the plot and not inserted into the table, it is now inserted into this newly active cell.)

The second problem of inserting a value into the text can be solved using the *DynamicTextSubstitution* module. Rather than simply using the HTML above, the instructor would have the following in the HTML document.

```
<a href=julia><replace value="exists(juliaC)"
                         true="@juliaC" false="?">%</a>
```

When the replacement is made on the entire document, if the "variable" `juliaC` has been set then it will be inserted. Otherwise, the '?' will be inserted before the % character.

How does the variable `juliaC` get set? To answer this, we first determine how information is retrieved from the plot when the user clicks on the axis. Each plot supports the listener paradigm via the *ToolListener* interface. We arrange that the *PostSampler* class implements this interface and register the object with the plot. Now, user interactions trigger a call to the method *toolModified()* which identifies the plot. We then query the value selected and compute the P-value.

Table 2: Input files required for an experiment in the hypothesis testing lab.

| Name | File type | Purpose |
|---|---|---|
| Icon | gif | image in the navigation bar |
| SelectionIcon | gif | image in the table of contents |
| Tooltip | HTML | tooltip for the icon |
| Description | HTML | text to accompany the selection icon |
| SubDescription | HTML | form for students opinion |
| Minimission | HTML | instructions to appear in the table of contents |
| Prestory | HTML | description of the experiment |
| | | to accompany the experiment animation |
| Animation | properties | instructions for the animation |
| Ticketfile | properties | ticket values for the sampler |
| Histogram | properties | description of histogram layout |
| Opinion | HTML | form for opinion table |
| Conclusions | HTML | form for final conclusion |

Figure 1: Topics screen for Hypotheis Testing

Now the information is available in the *PostSampler* object. We make these values available to the replacement process by assigning them to the *Context* object used in the replacement. We store the values under the name of the active expert's name suffixed by "C" for the critical-level and "P" for the $p$-value. This constrains the name of the links and the values specified in the HTML document but preserves the flexibility of the remainder of the content of the document and removes the need to parse the document in the code for this lab.

## 6.6 Extensions

To add an experiment to the lab, the instructor simply creates the appropriate input files (Table 2).

The instructor can also extend the lab to cover, for example, the concepts of alternative hypotheses and power. The first steps to doing this require consideration of the interface presented to the student. From a programming point of view, we should be able to use the existing classes with minor changes (e.g. adding new tab elements, buttons, etc.).

Figure 2: Experiment Animation for Hypotheis Testing

Figure 3: Sampler

Deborah Nolan
U.C. Berkeley
nolan@stat.Berkeley.edu

Duncan Temple Lang
Bell Laboratories
duncan@research.bell-labs.com

# References

[1] D. Freedman, R. Pisani, and R. Purves. *Statistics*. Norton, third edition, 1997.

[2] A. Gelman, D. Nolan with A. Men, S. Warmerdam, and M. Bautista. Student projects on statistical literacy and the media. *The American Statistician*, 1998.

[3] Information on SGML & XML.
URLs http://www.sil.org/sgml & http://www.w3.org.

Figure 4: Expert's opinions in Hypothesis Testing

Figure 5: Topics screen for Design of Experiments

Figure 6: Main work area for Design of Experiments

Figure 7: Virtual Experiment in Design of Experiments

Figure 8: Example Quiz Questions in Design of Experiments

Figure 9: Challenge round for Design of Experiments