DESIGNING TO FACILITATE CHANGE
WITH OBJECT-ORIENTED FRAMEWORKS

BY

BRIAN FOOTE

B.S., University of Illinois at Urbana-Champaign, 1977

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1988

Urbana, Illinois

DESIGNING TO FACILITATE CHANGE
WITH OBJECT-ORIENTED FRAMEWORKS

Brian Foote
Department of Computer Science
University of Illinois at Urbana-Champaign, 1988
Ralph E. Johnson, Advisor

Application domains that are characterized by rapidly changing software requirements pose challenges to the software designer that are distinctly different from those that must be addressed in more conventional application domains. This project assesses the impact of applying object-oriented programming tools and techniques to problems drawn from one such domain: that of realtime laboratory application programming. The project shows how a class inheritance hierarchy can promote the emergence of application specific frameworks as a family of applications evolves. Such frameworks offer significant advantages over conventional skeleton program-based approaches to the problems of managing families of applications. Particular attention is given to design issues that arise both during the initial design and implementation of such applications, and during later stages of their lifecycles, when these applications become the focus of reuse efforts. The project also addresses the impact of object-oriented approaches on the simulation of realtime systems and system components.

To Henry Hudson

Acknowledgements

This project has a longer history than most masters projects. This one is based on a larger system upon which I worked at the University of Illinois Department of Psychology's Cognitive Psychophysiology Laboratory (CPL). I am indebted to Emanuel Donchin, the CPL's Director, and to Earle F. Heffley III, the CPL's Technical Director, both for their willingness to allow me to explore new programming techniques with the CPL battery, which laid the foundation for this project, and for their support for my efforts to juggle school and work, which made this project possible.

I am grateful as well to Michael Faiman, Walter Schneider, and Arthur Kramer, who helped to convince me of the wisdom of pursuing an advanced degree.

Roy Campbell's advanced software engineering course had a major impact on my thinking about object-oriented framework lifecycle issues.

I'd like to thank Dan Ingalls, Kent Beck, and Barry Haynes of Apple Computer for providing me with up-to-date tools and assistance with the Macintosh Smalltalk that I used for the battery simulation.

Without the tireless help of my writing coach, Audrey Wells, this document would be in much sorrier shape than it is. The responsibility for those warts that remain, is of course, entirely mine.

Finally, I'd like to gratefully acknowledge the assistance, encouragement, enthusiasm and patience of my thesis advisor, Ralph Johnson. Without his willingness to embark on an open-end investigation rooted in an application domain somewhat foreign to day-to-day computer science, this effort (such as it is) would simply not have been possible.

Table of Contents

Chapter I -- Introduction

A good software designer knows that a job is not done when the requirements for the project at hand have been met. Well designed systems will lay the foundations for solving related problems as well. Good designers will always keep one eye open to opportunities to produce code and components that are usable beyond the scope of the current problem. A common result of such efforts is the accumulation of a library of broadly applicable utility components and routines. Another result can be the construction of robust, easily extensible applications that facilitate the evolution of a given application or component as the demands made of it change.

Not all programming systems and methodologies are equally effective in supporting the graceful evolution of applications and systems. Certainly high-level languages such as Algol-68 and Pascal, and approaches such as stepwise refinement and structured programming have done much to ease the burden of the programmer designing new applications. The notions of encapsulation and information hiding, which have been embodied in languages like Ada and Modula-2, have contributed much to our ability to deal with large, complex problems and systems. These approaches, powerful though they may be, only begin to address the sorts of pressures one faces when an existing system must adapt to new requirements. The central focus of the work described herein is on how object-oriented languages, techniques and tools confront the problem of volatile requirements.

This document describes a project (the *battery simulation*) that assesses the value of bringing object-oriented tools and techniques to bear on problems drawn from the domain of realtime laboratory programming. The project was based upon a large laboratory system (the *CPL battery*, or the *battery*) developed by the author using more traditional tools and approaches. The project involved the reimplementation of substantial portions of the CPL battery using Smalltalk-80. The principal question that motivated this effort was: How might the use of object-oriented tools and techniques affect the design, implementation, and evolution of programs in this application domain?

The problems of realtime laboratory programming are quite distinct from those in areas that have been more comprehensively studied by computer scientists, such as compiler construction or operating system design. Realtime laboratory data acquisition and experimental control applications must flourish in a research environment that is (by the very nature of research itself) characterized by rapidly changing requirements. These applications must also operate under severe timing and performance constraints, and must be designed to facilitate graceful evolution.

The applications which provided the basis for this project (the CPL battery) resulted from a fairly ambitious attempt to address some of the requirements stated above using traditional programming tools. A number of the approaches taken in the design of the CPL battery were inspired by object-oriented techniques. This project (the battery simulation) represented an attempt to ascertain what impact the use of a full-blown, bona-fide object-oriented programming environment (Smalltalk-80) might have on a redesign and reimplementation of representative portions of the CPL battery.

This Smalltalk-80 reimplementation had, from the onset, an exploratory character. An important aim of the project was to examine how the Smalltalk language and system might affect the sort of code produced to solve laboratory programming problems. The "plumbing" data stream classes for data analysis and the "accessable object" record/dictionary classes are two of the more interesting results. Another goal was to assess the utility of Smalltalk's user interface construction tools in constructing these applications. The waveform and parameter browsing tools incorporated into the project resulted from this effort. The plumbing data stream classes, accessable objects, and the battery browsing tools are presented in detail in Chapters III and IV of this document.

The overriding focus of this effort, however, was not as much to see how object-oriented techniques might aid in the construction of any given application as it was to assess the ways in which these techniques might be used to avoid the sort of wasteful duplication that is conventionally seen in these sorts of application domains <u>as</u> <u>requirements</u> <u>change</u>.

One way in which object-oriented schemes help to meet this goal is by encouraging the design of general, application independent libraries of reusable components. More conventional programming environments do this too. The information hiding capabilities present in object-oriented languages and systems are of great benefit in promoting the development of reusable libraries.

Another way in which object-oriented approaches facilitate graceful component evolution is via the ability of an object-oriented system to support the customization of a general kernel of components through the specialization ability provided by inheritance. The specialization and reuse capabilities provided by object-oriented inheritance and polymorphism increase the potential applicability of both preexisting and user generated system components. Hence, effort spent making a component more general is likely to pay off sooner than it might in a conventional system.

The Smalltalk battery simulation uses a class inheritance hierarchy to help manage a set of related, evolving applications as they diverge from a common ancestor. The emergence of an application specific framework in the face of volatile requirements is perhaps the most interesting consequence of the use of object-oriented techniques.

Traditional tools and approaches in the laboratory domain encourage a programming style built around a library of context independent reusable subroutines and disposable custom programs built (perhaps) from simple skeletons. An object-oriented approach allows a broad middle ground between these two extremes: the application framework.

The existence of a mechanism that allows the graceful evolution of program components from the specific to the general is a valuable asset during the design of any system, but the value of such a capability takes on an additional dimension of importance when such systems must evolve in the face of highly dynamic requirements. Thus, designing to facilitate change takes in lifecycle issues that normally are addressed under the rubrics of reuse, maintenance and evolution.

The Structure of this Document

This document is organized as follows:

Chapter II gives the history and background of this project, including a detailed description of the system upon which the project was based (the CPL battery), and discusses Smalltalk and object-oriented programming in general.

Chapters III and IV give the detailed anatomy of the Smalltalk-80 battery simulation framework and library.

Chapter V gives an illustrated tour of the battery simulation.

Chapter VI contains a discussion of a number of general questions and points raised by this research.

Chapter VII summarizes the project's results and conclusions.

Chapter II -- Background

This chapter describes the realtime laboratory application domain and the CPL battery, and discusses how the CPL battery spawned the Smalltalk battery simulation.

The Realtime Laboratory Application Domain

The Smalltalk battery simulation that is the focus of the effort described in this document was based on another system (the CPL battery) developed by the author while employed (as a systems programmer/analyst) at the University of Illinois Department of Psychology's Cognitive Psychophysiology Laboratory (CPL).

A basic psychophysiological paradigm employed by researchers at the CPL is the so-called "oddball" paradigm [Donchin 81]. In an oddball experiment, a subject (typically human) is presented with a Bernoulli series of one of two alternative stimuli, such as high or low tones. Each such presentation constitutes a <u>single</u> <u>trial</u>. A complete series of such trials is called a <u>block</u>. In the simplest case, the subject's task is to count each occurrence of one type of stimuli, and ignore all occurrences of the other.

During this procedure, EEG is collected from the subject via electrodes affixed to the subject's scalp. This EEG is amplified and and fed into a computer-driven analog-to-digital converter. The phenomena of interest are low amplitude signals that are difficult to detect in the wash of noise that is typically present in the EEG collected for any given single trial. Thus, signal averaging is necessary to produce aggregate responses across collections of many trials.

All aspects of data collection and experimental control are under the realtime control of a laboratory computer system. This system must concurrently generate and present stimuli to the subject, digitize and store collected data on some secondary storage medium, and conduct an interactive dialog with the experimenter, which may include the generation of realtime waveform displays and the presentation of statistical information.

A single trial might last for a period of 1 to 2 seconds. Data are typically digitized at 100 to 200 points per second. Data from 2 to 32 channels (electrode sites) are recorded for each trial, and stored in real time to magtape or cartridge tape. In addition to the analog data, digital input data reporting a subject response to a given stimulus, together with a response time (RT) may be collected for each trial.

The definition of what constitutes adequate realtime performance varies enormously from one application domain to the next. Certain applications in high energy physics may require that microsecond or better accuracy be present in the relative timing among experimental events. Many process control applications (and so-called realtime operating systems) effectively define realtime as being within a line clock "tick" (1/50th or 1/60th of a second). Psychophysiological applications usually require that experimental timing be accurate to within a millisecond or so.

The computer system used to run these experiments is one designed and constructed at the CPL. These systems, called Pearl II systems [Heffley 85] are built around DEC LSI-11/73 processors, and contain from 256k to 4M bytes of memory. Among the more noteworthy attributes of these systems is the set of 6 custom PC cards that provide 6 realtime clocks, a 16/32 channel 12 bit DMA clocked A/D system, a 4 channel clocked 12 bit DMA system, and a custom DMA cartridge interface.

Programs are developed and run under the DEC RT11 operating system. Application programs are usually written in Fortran IV, with the assistance of a structured preprocessor called FLECS [Beyer 75] that provides facilities such as advanced control structures, constant definition, and internal procedures that are not (or were not) present in the Fortran 66-based DEC Fortran IV implementation.

Application development is supported by a large library of Fortran and PDP11 assembly language (Macro-11) subroutines called LABPAK [Donchin 75], [Foote

85]. This library provides fast implementations of operations that are time or performance critical, or beyond what can easily be accomplished in Fortran (such as device drivers).

The LABPAK library hides much of the detail and complexity of dealing with realtime device programming and high performance data manipulation from the application programmer. Hence, laboratory application development can be undertaken by researchers themselves.

Naturally, there is a great deal of variation among the levels of programming skills exhibited by researchers in this environment. A typical simple laboratory application program might be a few hundred lines long. Usually, such applications are written by a single individual to solve some problem immediately at hand. (Often, the programs generated by researchers are variations on the oddball theme discussed above.) Once written, such an application program may take on a life of its own. This is because research, by its nature, ensures that no single application stays current for very long. The problems presented by moving target requirements will be addressed in detail in Chapter VI.

The CPL Battery

The CPL Battery was developed in response to requirements in several research contracts that called for the development of a battery of electrophysiological tests for the assessment of toxic exposure (for the EPA) and man-machine operator workload (for the Air Force). This package is discussed in a paper by Heffley, Foote, Mui and Donchin [Heffley 85]. The specifications for these tests called for the development of a battery of related applications that together would constitute a turn-key package with which a relatively unskilled operator could administer these tests. The specifications called as well for a great deal of latitude and flexibility in the configuration of these *battery items*. At the same time, each of the specified items was, in many respects, a variation on the oddball theme discussed previously.

It seemed obvious at the outset of this project that using the conventional approach of developing and maintaining a single separate application program for each required battery task was not feasible. The burden of maintaining such a large collection of programs appeared to have the potential of becoming overwhelming. The commonalities in the underlying structures of these applications ensured that large portions of each application would be duplicated across many or all of them. What's more, the high level of flexibility required by each of the specifications would have required that several versions of each item be developed if each had a scope similar to that of one of the applications we were used to.

Even when surface differences could be otherwise be reconciled, there was a need for variations in the user interfaces of the programs to tailor them to the needs of the various contractors, particularly with respect to the operator interfaces.

It was to meet these requirements that we developed the CPL Battery.

Typical CPL laboratory application programs have always had a brief list of parameters that could be modified by the experimenter using a simple table editing scheme.   This parameter dialog might allow an experimenter to alter between 10 and 50 experimental parameters. In order to allow the large number of variations called for in the specification, a much larger number parameters seemed necessary.

To support this capability, we developed a stand-alone parameter editor, which could accommodate up to 512 parameters per battery item. This editor is table driven, and provides both type and subrange checking. Integer, Option (Boolean) and String types are supported by the battery editor. The editor provided a full screen, arrow and key-letter driven interface (which was still a novelty during the early 1980s) as well as a per parameter help capability (which is still more of a novelty than it should be.) The editor also supports the realtime interpretation of symbolic arithmetic expressions, and included a primitive constraint resolution mechanism. It also provided facilities for storing and retrieving sets of parameter values.

The tables that drive the parameter editor are also the basis for the battery data management scheme. By retaining symbolic information describing the contents of battery data records, any battery program or other utility may access data generated by any other utility. These tables are generated by a preprocessor that takes a textual description of each parameter, along with its help information, and generates binary tables, help files, and FLECS common definition files. These include files provide an efficient mechanism for establishing a correspondence between symbolic names in the parameter editor and variables in the application programs themselves.

The development of support for a multi-page parameter dialog permitted the design of a handful of general applications that could, by virtue of the high degree of configurability made possible by the large number of parameters present, take the place of a large number of single purpose programs.

To attempt to exploit the structural similarities between battery items, a decision was made to use the internal procedure and conditional compilation capabilities of the FLECS preprocessor to attempt to mimic an object-oriented inheritance hierarchy. This attempt proved quite successful. Each of the eight or so battery items is derived from a single common FLECS framework. The sharing of large amounts of structural code among all the battery items permits one to lavish this common core with additional attention and features, with the knowledge that this effort will benefit all the applications derived from it.

This attempt to employ object-oriented principles was motivated by a keen interest on the part of the author in object-oriented programming, and Smalltalk in particular. It was this interest, combined with the battery effort, that motivated the Smalltalk battery simulation described herein.

A Tour of the CPL Battery

This chapter illustrates how an experimenter interacts with the CPL Battery. The screens depicted here show how an experimenter choses a battery item, inspects an item's parameters, and runs a block of trials.

```
CPL Test Battery (X03.00)

Tone (Auditory) Oddball
Visual (Box) Oddball Item
Word Oddball Program
Sternberg Test
Monitoring Task
Jex Critical Tracking Task
Brainstem Auditory EP Item
Pattern Reversal EP Item
Averaging Utility
Display (Measurement)
Global Parameters
EXIT

Use the VERTICAL ARROW Keys to Make your Selection.
Type RETURN when you are Finished.
```

The screen above shows the CPL battery's main menu page. Users select an item either by using the vertical arrow keys, or by using one of a set of single keystroke command synonyms.

Word Oddball

Run a Block of Trials

Inspect or Alter Parameters

Manipulate the Cartridge

Manipulate the Magtape

EXIT

```
Use the VERTICAL ARROW Keys to Make your Selection.
Type RETURN when you are Finished.
```

The screen above shows the main menu of the Word Oddball battery item. User may run a block of trials, inspect and change parameter values, or manipulate and examine the contents of either a Pearl II 1/4 inch cartridge tape, or a 9 track magtape.

Word Oddball

```
Page Name:  User ID Directory                                      Page 1

Page ----> : Data Parameters              Page ----> : Trial Parameters
Page ----> : Block Parameters             Page ----> : Data Control
Page ----> : Movement Rejection           Page ----> : Clipping Control
Page ----> : External Triggering          Page ----> : Stimulus Generation
Page ----> : Warning Control              Page ----> : Warning Words
Page ----> : Digital Outputs              Page ----> : D/A Tone Control
Page ----> : Matrox Control               Page ----> : Word Parameters

              Type H for HELP, Q to Quit this Section

ID Name     Type       Value     Id/Offset   Access       Low     High
-------     Direct     -----     (2) 2       Constant     ---     ----
```

The screen above depicts one of the Word Oddball items parameter directory pages. This battery item has three such pages. Each entry in the directory refers to a parameter page that may in turn contain up to 16 parameters.

```
                         Word Oddball

  Page Name:  Data Parameters                                      Page 4

  Calc Epoch =   Yes   [Y/N]            Epoch      =     995  MilliSec
  Calc Rate  =   No    [Y/N]            A/D Rate   =    5000  MicroSec
  Calc Pnts  =   No    [Y/N]            Points     =     200  #
  Start Chan =      1  [1-16]           Channels   =       5  [1-16]
  Bytes Used :   2000  Bytes            A/D Gain   =       1  [0-3]
  Bytes Free :  23600  Bytes            Max Bytes  :   25600  Bytes
  Ms. Base   :   No    [Y/N]            Dictionary : WO1013


                 Type H for HELP, Q to Quit this Section

  ID Name      Type        Value     Id/Offset   Access       Low     High
  A/D Gain     Integer         1     (48) 48     User           0        3

              A/D Gain -- Digitizer programmable gain setting.
       The appropriate value is determined by the output of the EEG
    amplifiers being used.  The values:  0:  -10 to 10V, 1: -5 to 5V,
                 2:  -2.5 to 2.5V, 3:  -1.25 to 1.25V.
```

This screen shows one of the Word Oddball's parameter pages. This page contains parameters associated with data collection. The values of the A/D digitizing rate, the A/D digitizing time, and the number of data points per sweep may be traded off against one another. The results of doing so are immediately displayed. The single degree of freedom present in this calculation may be assigned using the "Calc" parameters. The effect of such manipulations on memory usage is reflected immediately in the "Bytes Free" parameter. Such calculations are conducted by special constraint satisfaction code built into the menu management program that drives this display.

The text at the bottom gives a brief description of the "A/D Gain" parameter. Such interactive help is available for all of a battery item's parameters. The user alters a parameter by moving the cursor to it using arrow keys, and typing return. The user may then type an expression, the value of which becomes the new parameter value, if the parameter's limit constraints are not violated. This expression may include most standard Fortran and C operators, as well as symbolic parameter names. Hence, "A/D Gain"*2+1 is a legal expression.

The Battery main menu and parameter dialogs seen here are all conducted by a single program, called MENU. This program operates on tables produced by a preprocessor called DICT. The input to this program is an ASCII description of each parameter, giving its type, limits, and initial value. The help information is also given in this file. The DICT program produces tables for the MENU program, along with Fortran common and constant definition files that are used by the FLECS preprocessor to allow efficient Fortran level access to battery variables.

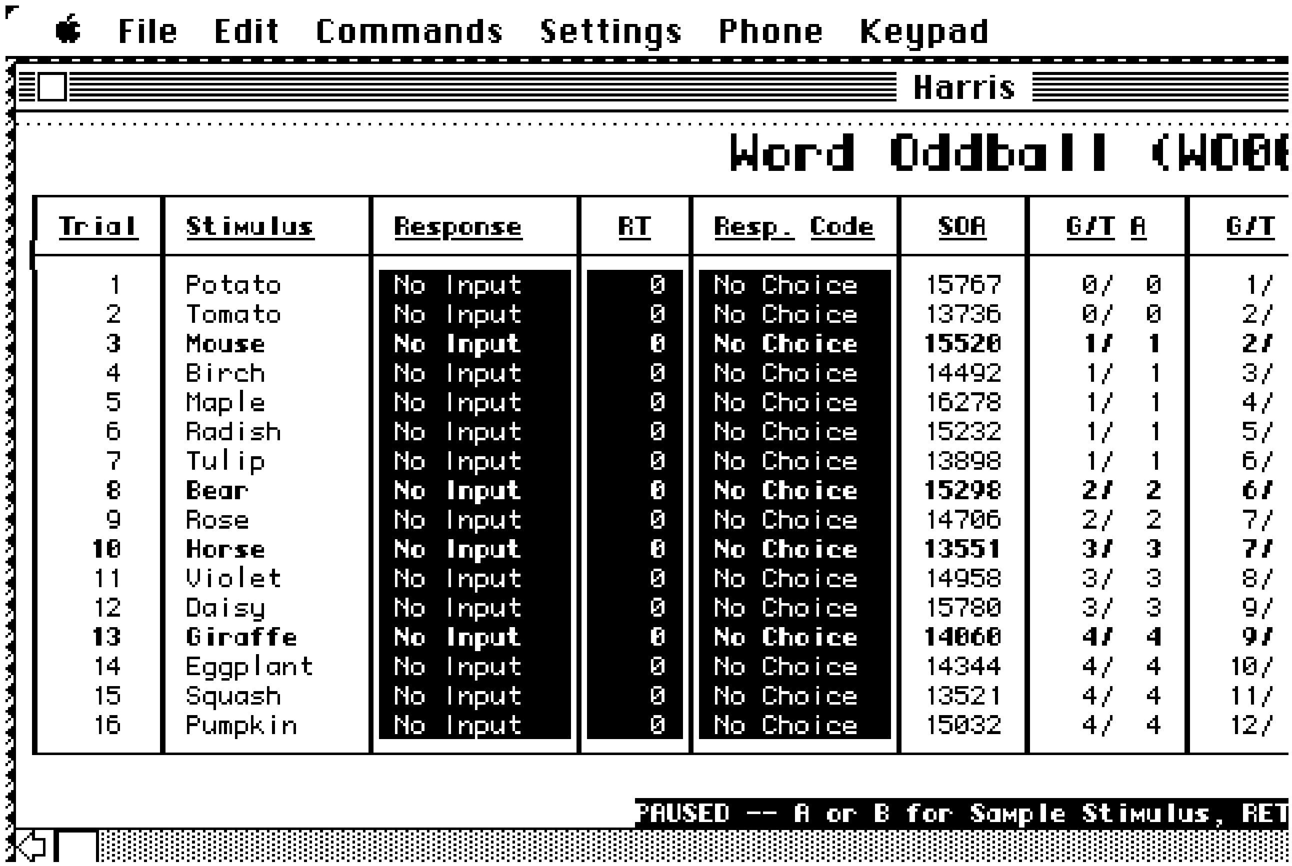
```
                         Word Oddball

                    Parameter Page Commands

Arrow Keys   --  Move Selector            F  --  Forward to Next Page
R            --  Read Parameters          B  --  Back to Previous Page
W            --  Write Parameters         H  --  For this HELP Page
Y or N       --  Change an Option         D  --  Describe Selection
# Return     --  Move to Given Page       S  --  Search for Parameter
+/- # Return --  Skip Pages               Q  --  Quit the Parameter Dialog

            RETURN will let you Alter your Current Selection
                      or Choose a Directory Item

                       Type RETURN to Continue

            A/D Gain -- Digitizer programmable gain setting.
        The appropriate value is determined by the output of the EEG
    amplifiers being used.  The values:  0:  -10 to 10V, 1: -5 to 5V,
                 2:  -2.5 to 2.5V, 3:  -1.25 to 1.25V.
```

The screen above gives a list of the commands available to the user in the parameter editor.

Block Preview Information

```
Epoch Length:  995 Milliseconds.  Baseline:  95 Milliseconds.
5 Channels, 200 Points/Chan Digitized every 5000 Microseconds.
2000 out of an available 25600 Bytes/Trial will be Used.

Stimuli are Program Generated.
Trials:  100, 21 Category A Trials (21%), and 79 Category B Trials (79%).
Anticipated Block Duration:  3:20.
SOA will vary between 1800 and 2200 Milliseconds.

Stimuli will be delivered through D/A Unit 2, Channel 2.
Category A Intensity:  1024 Units.  Category B Intensity:  1024 Units.

Category A Stimuli on Output Bit 0, Category B Stimuli on Bit 1.

Visual Stimuli will be presented on the Matrox Display.



                    Q to Quit, RETURN to Continue
```

Thescreen above shows the information given to the experimenter just before a block is begun.



The Macintosh screen dump above shows part of the Word Oddball items runtime display. Trial categories are distinguished by bold and dim text. The trial number is given in the leftmost column, followed by the stimulus. The next three columns give subject response information. Since no subject was present when this example was run, the reaction time column (RT) is filled with zeros, and the response columns are written in reverse video, indicating that the responses for this trial are in error.

Why a Smalltalk Battery Simulation?

A number of factors made simulating the CPL battery in Smalltalk seem like a potentially worthwhile enterprise. The CPL battery was constructed using techniques and approaches that were taken from Smalltalk. The simulated inheritance hierarchy used to manage related applications and the highly interactive user interface present in the CPL battery were both inspired by

Smalltalk. Both were simulated at considerable expense in the CPL battery. It seemed natural to ask what advantage there might be in using Smalltalk itself to construct a system with similar objectives.

The inheritance mechanism in Smalltalk is much more powerful (and less coarse) than the one used in the CPL battery. As for the user interface, Smalltalk is (rightfully) well known for having originated the bit-mapped, windowed user interface style that is so popular today. The Smalltalk environment comes with an extensive array of classes that support this user interface.

It seemed interesting to ask as well what impact an object-oriented language like Smalltalk might have on low level coding. Object-oriented polymorphism seemed to have the potential to allow low level code to cope with the needs of a family of related applications in a tidier fashion than conventional languages (like Fortran) might allow. Also, object-oriented programming styles encourage the development of a collection of modular application specific abstractions. Assessing the impact of using such abstractions on the coding process seemed a worthwhile pursuit.

Above all, however, the battery simulation was inspired by a desire to investigate the impact of object-oriented techniques and environments on the design of a family of diverging applications. In particular, I wanted to determine how a Smalltalk implementation of a system like the battery would support the sort of extension and change present in a laboratory environment.

Certain other aspects of the Smalltalk system made it attractive as well. Under Smalltalk-80, all the objects known to a given copy of the system are resident in that system's image. It struck me that such an environment might be an interesting place to investigate new ideas about managing the large volumes of data that are produced by laboratory applications, particularly with respect the problem of identifying and labeling such data. In a resident environment, stored data could retain their original identities as objects, and respond to their full repertoire of messages. The combination of resident data and the Smalltalk user interface tools seemed to have considerable potential.

One last aspect of Smalltalk versus the CPL development environment seemed compelling as well. The Apple Lisa and Macintosh MC68000 systems that supported the Smalltalk battery simulation had up to 2 megabytes of directly addressable memory available to them. The original CPL battery ran (and still runs) on DEC PDP-11 family processors that can, for most purposes, address only 64kb of memory directly.

Chapter III -- Anatomy of the Battery Framework

The Battery simulation is an object-oriented application framework for constructing psychophysiological experiments of the sort described in chapter II. Experimenters add new experiments to this framework by adding new subclasses to this framework. The number of new subclasses that must be added and the amount of new code that must be written will depend (of course) on the degree of similarity between the requirements for the new experiment and parts of the existing framework.

This chapter, and the one that follows it, give a detailed exposition of all the code in the Battery simulation. It is hoped that the battery simulation can serve as a case study in the application of object-oriented frameworks. Every line of code in the simulation is presented, warts and all. The class definitions for this code are organized into 14 major categories. (The names given are Smalltalk system category names.) The first 8 categories contain code that is relatively battery specific. These constitute the bulk of the battery framework. The remaining 6 categories contain code that is largely battery independent. These classes hence play a role similar to that of library components in a conventional system. These classes are discussed in Chapter IV. The categories are:

Battery-Items	The Sternberg, Word and Tone Oddball
Battery-Parameters	Parameter carriers for the Battery Items
Stimulus-Generators	Pluggable stimulus presentation devices
Stimulus-Support	Stimulus descriptor objects
Sequence-Support	Stimulus sequences generation support
Response-Support	Subject response descriptor objects
Data-Management	Data archive simulation objects
Interface-Battery	Support for browsing battery items
Realtime-Support	Simulated realtime timebase code
Realtime-Devices	Clocks, digitizers, digital I/O devices
Waveform-Support	Waveform collection and statistics
Random-Support	Random integer and stream support
Plumbing-Support	Objects for connecting streams together
Accessable-Objects	Support for record/dict. transparency

The Battery-Items category contains the core of the battery simulation: the abstract class BatteryItem. BatteryItem is the principal component of the psychophysiological experimental application framework. This class is the superclass of the three concrete battery items presented here: WordOddball, ToneOddball, and SternbergTask. One of the central goals of this simulation was to demonstrate how a class inheritance hierarchy could be used allow a family of related application programs to share common code.

The Battery-Parameters category contains a set of record-like objects that carry the user modifiable parameter values for each battery item. These are organized in a hierarchy that mirrors the battery item hierarchy. The use of separate objects to embody the configurable state of battery items simplifies and isolates the design of the battery parameter viewing mechanism.

The Stimulus-Generator category embodies an attempt to construct interchangeable stimulus generation modules that could be fitted into any appropriate application.

The Stimulus-Support and Response-Support categories contain classes that function as descriptors that ferry information between the battery items themselves and the stimulus generation and response reporting systems.

The Battery-Parameters, Stimulus-Generator, Stimulus-Support, and Response-Support categories are examples of objects which have been factored out of the original battery item design as discrete components. An experimenter constructing a new experiment will extend BatteryItem, or some subclass thereof, along with members of these hierarchies. Because these elements are not subclasses of BatteryItem itself, these components can be mixed and matched as black boxes among BatteryItems.

The Sequence-Support category supports the generation of experimental stimulus sequences.

The Data-Management category contains a set of descriptor classes that provide the foundation for a dictionary-based data archiving scheme. Class BatteryTrial contains waveform data and descriptive scalar information for each single trial that is performed. A BatteryBlock contains all the trials collected during a given experimental run. Each completed BatteryBlock is entered into a

DataDictionaryEntry descriptor and entered into a DataDictionary. Class BatteryItem contains an instance of DataDictionary that serves as the master repository for battery data.

The Interface-Battery category supports the Battery browser. This browser allows an experimenter to access a list of battery items, inspect or alter the parameters for a given battery item, and run a battery item. A running battery item maintains a dynamic waveform display.

The code for the battery independent categories is described in the Chapter IV.

The code for each system category is presented in turn. A general description of each category is given first, along with a list of the classes defined in that category. Each class in the category is then presented. Each class description begins with a table that resembles a standard Smalltalk class definition template. The class name and its superclass are given, first followed by lists of instance and class variables. Inherited instance variable names are given in *italics*. These lists are followed by the pool dictionary list, the category name, and a table showing the class hierarchy.

Battery-Items

The classes in the Battery-Items category constitute the framework around which the Battery simulation is built. The classes are:

BatteryItem	A generic battery item
SternbergTask	A Sternberg memory task
ToneOddball	An auditory oddball experiment
WordOddball	A visual semantic oddball experiment

BatteryItem

Class BatteryItem is the central core of each of the three actual battery items implemented herein. BatteryItem is an abstract superclass, since it exists only to provide a repository for the common behaviors among the three concrete battery items.

A BatteryItem is a generic psychophysiological experiment. A BatteryItem object's job is to run a block of experimental single trials whenever it receives the **doBlock** message. A *block* is a collection of a designated number of single trials. The number of trials in a block is an experimental parameter.

A *single trial*, or trial, corresponds to the presentation of a stimulus or a sequence of stimuli to an experimental subject. For each trial, a BatteryItem must coordinate the timing and presentation of stimuli, the collection of A/D waveform data (using an analog-to-digital converter, or digitizer), the collection of discrete response time information, and the maintenance of cumulative statistics and experimenter displays. A BatteryItem must accurately orchestrate all these activities in (simulated) real time.

BatteryItem is a subclass of AccessableObject, which enables each of its instance variables to be accessed as though a BatteryItem were a dictionary. AccessableObjects allow new key/value pairs to be defined, which will then also respond to the conventional instance variable accessing protocol (**item**, **item: value**). See the discussion of category Accessable-Objects for more information on what AccessableObjects can do. BatteryItems inherit the instance variable *items* from AccessableObject. This instance variable helps to provide the soft instance variable facility mentioned above.

The class variable *BatteryDataDictionary* is an important component of the battery simulation's rudimentary data management scheme. Rather than writing battery data to some external medium, the battery simulation enters the data collected during each run of a battery item into a dictionary that is kept in *BatteryDataDictionary*.

class name BatteryItem
superclass AccessableObject
instance variable names

items

trialClock
stimulusClock
digitizer
responseDevice
parameters
stimulusSequence
sequenceGenerator
stimulusGenerator
buffer
experimenterLog
waveformDisplay
block
blockNumber
nextTrial
currentTrial
currentTrialNumber
outputStream
averagePipeline
tallyPipeline

class variable names

BatteryDataDictionary

category Battery-Items
hierarchy

```
Object
    AccessableObject
        BatteryItem
            SternbergTask
            ToneOddball
            WordOddball
```

BatteryItem methods for: block control

The block control methods control block preparations, the sequencing of single trials, and block termination activities.

The fundamental operation that a BatteryItem can be called upon to perform is to **doBlock**, which defines the basic sequence of block activities. Any time consuming block preparations are made by **prepareBlock.** These include updating the block counter, preparing a stimulus sequence, preparing the data management system, and setting up the dataflow data processing fixtures.

Then, time-critical block start-up actions are performed by **startBlock**, such as informing the stimulus generator object that the block is starting.

The division of labor among the components that constitute a battery item is such that each battery item works with a discrete stimulus generating object of some sort. For some applications, the internal mechanisms of this object might be quite complex, and hence there is a need to synchronize it with the block and trial sequencing done by the battery item.

The **runBlock** method is the heart of the block control mechanism. It zeros a trial counter, and proceeds to generate and execute trials. A trial is executed by sending the **doTrial** message to a battery item. The trial control code is discussed below.

When the block has completed, the **finishBlock** method is executed. This method tells the data management and stimulus generation components to perform any block termination activities that they might require.

An experiment that required that basic changes be made to a BatteryItem's block sequencing behavior might override these methods. For example, an experiment that required that a sequence of blocks be run in order with only minor, systematic changes to experimental parameters might override some of these methods.

BatteryItem methods for: block control

doBlock

"To run a block, we must make adequate preparations, start per block activity, run the blocks's trials themselves, and finish up. The fundamental service a BatteryItem provides to the outside world is the ability to do a block when it is requested to."

```
self prepareBlock.
self startBlock.
self runBlock.
self finishBlock
```

finishBlock

"We are done. Clean up. We perform whatever end-of-block activity our data management scheme may require, and tell our stimulus generator that the block is over."

```
self finishDataManagement.
```

```
self stimulusGenerator finishBlock
```

prepareBlock

```
"Block preparations are those time-consuming tasks that need to be
performed before a block is started.  These are as opposed to actions of an
essentially instantaneous nature, which should be enumerated in
startBlock. "

self blockNumber: self blockNumber + 1.
self prepareStimulusSequence.
self prepareDataManagement.
self prepareProcessingFixtures
```

runBlock

```
"To actually run the block, we have to generate a bunch of trials.  We
overlap the completion of the current trial and the preparation of the
next in doTrial.  Hence, we must prime this process here."

self currentTrialNumber: 0.
self prepareNextTrial.
(1 to: self trials)
    do:
        [:n |
        self currentTrialNumber: n.
        self currentTrial: self nextTrial.
        self doTrial]
```

startBlock

```
"Tell our stimulus generator we are starting up."

stimulusGenerator startBlock
```

BatteryItem methods for: trial control

The trial control methods are templates for a BatteryItem's basic trial control sequencing.

The **doTrial** method conducts an experimental trial. The **startTrial** method starts the single trial timer (which is returned by the **trialClock** method) for an amount of time indicated by the **trialDuration** method. The **runTrial** method is the generic trial body. It calls **collectData** and **processData.**

The **collectData** method must be provided by BatteryItem's subclasses, and is, in this implementation of the battery, the primary locus of differences among BatteryItem's concrete subclasses.

The **processData** method takes the data deposited in the buffer object and per-trial data structures and performs per-trial processing on them.

The **prepareNextTrial** method constructs the per-trial data objects for the next trial. This method allows us to overlap the perhaps time consuming preparation of the next trial with the often relatively quiescent period at the end of the current trial.

The **finishTrial** method simply waits for the trial clock's time to elapse.

BatteryItem methods for: trial control

doTrial

```
"To do a trial, do the trial start up, run the trial, prepare the next one,
and clean up.  We prepare the next trial before waiting for the time
allotted for the current trial to elapse (which is done in finishTrial)."

self startTrial.
self runTrial.
self prepareNextTrial.
self finishTrial
```

finishTrial

```
"We'll always end by waiting for the trial clock.  In the interest of tidy
timing, subclasses that override this method might want to send super
finishTrial last."

trialClock wait
```

prepareNextTrial

```
"Do stimulus selection and preparation here.  We allocate a BatteryTrial
object, and set it up."

self nextTrial: BatteryTrial new.
self nextTrial trialNumber: self currentTrialNumber + 1.
self currentTrialNumber < self trials
    ifTrue:
        [self selectNextStimulus.
        self prepareNextStimulus]
```

runTrial

```
"This is our generic trial body.  Each trial first collects some data,
then processes and stores them."

self collectData.
self processData
```

startTrial

```
"We'll always start by starting up the trial clock.  In the interest of tidy
```

timing, subclasses that override this method might want to send super startTrial first."

```
self trialClock startFor: self trialDuration
```

BatteryItem methods for: instance initialization

The methods in this category allocate battery item resources and set default values.

The methods in this category are all called in response to an **initialize** message to a BatteryItem. The **initialize** message zeros the block number instance variable, and sends an **allocateResources** message to the BatteryItem. This method then invokes the remainder of the resource allocation code. They are concerned primarily with the one-time creation or allocation of per-item resources.

Note that **allocateParameters** and **allocateStimulusGenerator** are responsibilities of the concrete subclasses of BatteryItem. None of the other initialization messages in this incarnation of the battery are overridden.

BatteryItem methods for: instance initialization

allocateBuffers

"Allocate the buffer our A/D system will use. The A/D system fills a pre-existing buffer rather than creating and returning every time it is asked to collect data. The buffer: message stores this buffer somewhere until it is needed."

```
self buffer: (WaveformCollection channels: self channels points: self points)
```

allocateClocks

"We will use two timers, a trial timer, and a stimulus presentation timer. We allocate and store these here."

```
self trialClock: Clock new.
self stimulusClock: Clock new
```

allocateDigitizer

"Let's use a buffered digitizer for now. (There is a StreamedDigitizer as well.) Store this somewhere where we can get at it."

```
self digitizer: BufferedDigitizer new
```

allocateParameters

"Let our subclasses deal with this..."

```
self subclassResponsibility
```

```
allocateResources
    "Create the resources we will be employing to conduct this experiment,
    such as parameter blocks, clocks, buffers, digitizers, stimulus generators,
    sequence generators, and response devices."

    self allocateParameters.
    self allocateClocks.
    self allocateBuffers.
    self allocateDigitizer.
    self allocateStimulusGenerator.
    self allocateSequenceGenerator.
    self allocateResponseDevice

allocateResponseDevice
    "Build a button box and save it somewhere."

    self responseDevice:
        (ButtonBox bitA: self inputBitA bitB: self inputBitB usingClock: self stimulusClock)

allocateSequenceGenerator
    "Allocate a sequence generator."

    self sequenceGenerator: SequenceGenerator new

allocateStimulusGenerator
    "Let our subclasses worry about this..."

    self subclassResponsibility

initialize
    "Create our block resources, and zero the block counter."

    self blockNumber: 0.
    self allocateResources
```

BatteryItem methods for: data collection

Data collection strategies will usually differ sufficiently among different battery items so as to make it necessary for this code to be implemented further down in the inheritance hierarchy.

The **collectData** method must be implemented by each subclass of BatteryItem to orchestrate the basic data collection sequence for that item. This method will usually contain a sequence of experimental control and data acquisition method invocations that will together define the heart of the experiment implemented by a given battery item.

The **startDigitizer** method will often (always given the current framework) be called from **collectData** to start the A/D system.

BatteryItem methods for: data collection

```
collectData
    "Data collection requirements differ from item to item, so our subclasses
    must implement this."

    self subclassResponsibility

startDigitizer
    "Start our A/D converter.  We ask ourself for the various parameters we
    need to start this process.  We care not from where these values actually
    come... "

    self digitizer
        collectChannels: self channels
        points: self points
        in: self buffer
        every: self digitizingRate
        thenDo: []
```

BatteryItem methods for: data processing

This protocol category contains a set of generic methods for setting up and processing collected data.

The **prepareProcessingFixtures** method prepares the data averaging and statistical data accumulation pipelines. The averaging fixture set up method, **prepareAverageFixtures** should be provided by subclasses of BatteryItem. Each subclass will typically call **average:using:withLabel:** for each average category it wishes to define.

The **average:using:withLabel:** method constructs a dataflow pipeline that evaluates a given source block after the data for each trial are collected, and forces the result of that block down a pipeline using the indicated valve (conditional) block. If this valve is open, this result can then flow into the Averager object for the pipeline. The pipeline is constructed using the pseudo-Stream objects defined in the system category Plumbing-Supplies.

The **average:using:withLabel:** method works as follows: First, an average buffer into which a running average will be accumulated is created. Next, this buffer (the WaveformCollection) is entered into the average dictionary in the current TrialBlock object using the given label as a key. Next, a Pump object is created, and a ValueSupply object using the given source block as its source is connected to the input (source) side of the Pump (using the **<<** or connect input message). Then, the output side of the dataflow pipeline is connected (using **>>**) to the effluent (sink) side of the pump. The first stage of the output pipeline is the Valve object, and the final stage is an Averager on the average buffer we created. Finally, the fixture is entered into an averagePipeline dictionary kept by this instance of BatteryItem.

The **tally:using:withLabel:** method uses a similar scheme to construct pipelines for incrementally accumulating statistical information using Tally objects. Tally objects are objects into which pipelined data can be fed that monitor the data that pass through them and accumulate statistical information about those data. Tally objects can later be asked for statistical characterizations of the data that have passed through them. (For instance, they can be asked to give a count, or an average, of the data they have seen.) The nature of the data collected will of course depend on the nature of the Tally object itself. The pipelined scheme used here allows the nature of strategies for accumulating and processing statistical data to be implemented independently of the contexts in which they are employed.

The **prepareStatsFixtures** method constructs a set of fixtures for collecting BatteryItem button box response time data. First, the tally pipeline and tally collection dictionaries are created. The tally pipeline dictionary holds the root of each pipeline we have built (i.e. its Pump object), so that when the time is right, each pipeline can be told to move data from its ValueSupply block down (Value permitting) to its effluent sink (the Tally object). The TrialBlock's tally dictionary provides access to the Tally objects at the end of the pipelines for posterity's sake. The pipeline fixtures and tally objects are entered into these dictionaries by **tally:using:withLabel:**. The real work in the **prepareStatsFixtures** method is performed by calls to **tally:using:withLabel:.**

For all three of the pipelines constructed by this method, the values are generated using a block that asks the current trial for its response time. The pipelines differ in the labels given them, and in the condition under which the pipeline's Valve object will permit the data returned as a result of evaluating the value block to be passed through to the Tally object. The first pipeline checks only that the value passed it is not *nil*. Each of the other two adds at test for whether one or the other of the response buttons was pressed in addition to the test for *nil*. Note that the block passed through to the ValueSuppy takes no arguments, while the block passed to the Valve object takes one argument. When a Valve object is passed a value, it will pass this value in turn to its test block as the block's argument.

The purpose of all of this is to simplify the process of adding new statistical categories to data collection programs. Using the pipelining scheme, all one would have to do to add a new statistical category is make a single additional call to **tally:using:withLabel:** to define the new category. All of the statistical characterizations of which the Tally objects are capable will then be available. It is interesting to contrast the relative ease of this process with the tangle of additional variables, declarations and code that is typically generated when an experimenter desires that new statistical categories be added to a program using a conventional programming language, such as Fortran. The pipeline approach encapsulates the details of the statistical collection process, such as the temporary variables, the category criteria, etc.

The **processData** method is called from **runTrial.** It outlines our generic mechanism for dealing with collected data. First, the subject's response, (if any) is evaluated. Next, the data for this trial are stored. Then the per-block averages and statistics are updated. Finally, the waveform and experimenter log displays are refreshed.

The system transcript is used as an experimenter log in this implementation of the battery. Single channel number one of the (single trial) data buffer is displayed in the BatteryItem's WaveformView.

The **updateAverages** and **updateStatistics** methods illustrate how data are extracted from the ValueSupply blocks of the plumbing fixtures and are flushed down into the effluent sides of the pipelines. Each pump object in the average and tally pipeline dictionaries is sent the message **nextPut** to accomplish this. The **nextPut** method moves one value from the intake to the effluent side of a Pump.

BatteryItem methods for: data processing

```
average: sourceBlock using: valveBlock withLabel: label
    "Build a fixture dictionary entry."

    | fixture averageBuffer |
    averageBuffer ← WaveformCollection channels: self channels points: self points.
    self block averages at: label put: averageBuffer.
    fixture ← Pump new.
    fixture << (ValueSupply using: sourceBlock).
    fixture >> (Valve using: valveBlock) >> (Averager on: averageBuffer).
    self averagePipeline at: label put: fixture

prepareAverageFixtures
    "Let our subclasses deal with this..."

    self subclassResponsibility

prepareProcessingFixtures
    "Prepare the tally and average pipelines."

    self prepareAverageFixtures; prepareStatsFixtures

prepareStatsFixtures
    "Prepare the tally pipelines."

    self tallyPipeline: (Dictionary new: 10).
    self block tallies: (Dictionary new: 10).
    self
        tally: [self currentTrial responseTime]
        using: [:rt | self currentTrial responseTime ~~ nil]
        withLabel: 'RT Statistics (All trials)'.
    self
        tally: [self currentTrial responseTime]
        using: [:rt | self currentTrial responseTime ~~ nil and:
            [self currentTrial responseType = #buttonA]]
        withLabel: 'RT Statistics (Button A)'.
    self
        tally: [self currentTrial responseTime]
        using: [:rt | self currentTrial responseTime ~~ nil and:
            [self currentTrial responseType = #buttonB]]
        withLabel: 'RT Statistics (Button B)'

processData
    "Data collection is complete.  We must now evaluate the digitized data
    and the subject's response (if any), and update the experimenter's
```

waveform and log displays."

```
self evaluateResponse.
self storeCurrentTrial.
self updateAverages.
self updateStatistics.
self updateWaveformDisplays.
self updateExperimentalLog
```

tally: sourceBlock using: valveBlock withLabel: label

```
"Build a fixture dictionary entry."

| fixture tally |
tally ← Tally new.
self block tallies at: label put: tally.
fixture ← Pump new.
fixture << (ValueSupply using: sourceBlock).
fixture >> (Valve using: valveBlock) >> tally.
self tallyPipeline at: label put: fixture
```

updateAverages

```
"Give our pipeline pumps a little push..."

self averagePipeline do: [:fixture | fixture nextPut]
```

updateExperimentalLog

```
"Write a message for each trial.  We use the System Transcript as the log
medium. "

Transcript show: self trialNumber printString , ') '.
Transcript show: 'Stim: ' , self stimulus category printString , ' '.
Transcript show: 'Resp: ' , self responseType printString , ' '.
Transcript show: self responseTime printString , ' ticks'; cr.
Transcript endEntry
```

updateStatistics

```
"Give our pipeline pumps a little push."

self tallyPipeline do: [:fixture | fixture nextPut]
```

updateWaveformDisplays

```
"Show a single trial channel for now.  FLECS battery items typically
show several single trial and average channels.  First, ask our
parameter's object for the battery browser object we are to use.  Then, ask
it for its WaveformView.  Change the model for this view to our first
channel (an arbitrary choice, I admit) and change the view title to
indicate the current trial number.  Then, send this view a update:
#waveform message"

| aBrowser aWaveformView |
aBrowser ← self parameters browser.
aWaveformView ← aBrowser waveformView.
aWaveformView model: (self buffer at: 1).
aWaveformView title: 'Trial ' , self trialNumber printString.
aWaveformView update: #waveform
```

BatteryItem methods for: data management

The methods in this category implement a generic data management scheme.

This implementation of the battery uses a class variable common to all instances of BatteryItem to store a global data dictionary. The **dataDictionary** method returns this dictionary.

Every time a battery item collects a block of data, it enters a DataDictionaryEntry object containing the TrialBlock object into the data dictionary under a name generated by the **entryName** method. The **entryLabel** method provides additional descriptive information. This label is stored in the DataDictionaryEntry object as well.

The **prepareDataManagement** method allocates a new BatteryBlock object, and fills it with a copy of the BatteryItem's own parameter object, and an empty trial array. It then creates a WriteStream over the trial array, and installs this as the BatteryItem's output stream. To the rest of the BatteryItem code, outputStream is merely a sink of some sort down which single trial objects can flushed.

The **storeCurrentTrial** method copies data from the digitizing buffer into the current trial object, and sends this object down our outputStream.

The **finishDataManagment** method is charged with creating the DataDictionaryEntry for a completed block, and storing this entry in the master data dictionary.

BatteryItem methods for: data management

```
dataDictionary
    "Return our data dictionary.  This is a class variable shared by all
    BatteryItems into which the result of individual battery runs are
    entered. "

    ^BatteryDataDictionary
```

entryLabel

"Just use the item label. Each entry we make into the BatteryDataDictionary will have an entryLabel and an entryName."

```
^self label
```

entryName

"Return the name parameter with the block number appended. Each entry we make into the BatteryDataDictionary will have an entryLabel and an entryName."

```
^self name , ' #' , self blockNumber printString
```

finishDataManagement

"If we got this far, we'll presume that it's okay to enter our data in the data dictionary. We do so here. Note that we could just as easily be closing a data file or a data stream, should an alternate data management strategy require this. (Note that from the perspective of the rest of the BatteryItem code, the data management scheme appears as one in which single trial objects are sent down a WriteStream-like sink.)"

```
| anEntry aName |
aName ← self entryName.
anEntry ← DataDictionaryEntry
    entryNamed: aName
    withLabel: self entryLabel
    andData: self block.
BatteryDataDictionary at: aName put: anEntry
```

prepareDataManagement

"Allocate a block data object, and set up a stream over it. To the bulk of our code, the data management scheme will appear as one that requires single trials to be sent down some sort of outputStream. The code will care not whether this is a file, a collection, or what have you..."

```
| trialArray |
block ← BatteryBlock new.
block parameters: self parameters shallowCopy.
trialArray ← Array new: self trials.
block singleTrials: trialArray.
self outputStream: (WriteStream on: trialArray)
```

storeCurrentTrial

"Copy the digitized data, and set the current trial on its way down whatever it is we are using as an outputStream..."

```
self currentTrial data: self buffer deepCopy.
self outputStream nextPut: currentTrial
```

BatteryItem methods for: stimulus control

The battery simulation uses discrete stimulus presentation objects to generate stimulus sequences and present stimuli. The stimulus control protocol coordinates the various objects involved in stimulus generation.

The **prepareStimulusSequence** method sets up a stream from which individual stimuli can be read for each trial. It does this by first creating a sequence (using the sequenceGenerator) using a given collection of stimulus template objects, a trial count, and a collection of relative stimulus probabilities. We then set up a SampledStream over this sequence so that we may draw from it in a random order without replacement. The sequence generation scheme is interesting in that it is constructed using parts (WeightedCollections and SampledStreams) that were much more specific to this application domain earlier in the design process. The current design broke out these two potentially general components and employs the Unix philosophy of constructing special purpose components by composing a handful of general building blocks rather than building a single special purpose component from scratch.

The **selectNextStimulus** method merely copies the next stimulus from the stimulus stream into the current trial object. Note that we make a deepCopy of the stimulus object rather than passing a reference to it. This is because the stimulus sequence uses multiple references (via the WeightedCollection) rather than copies, and because there is no telling whether a subsequent analysis application might elect to make some destructive reference to one of these objects. (The issue of when to copy an object or when to pass a reference to it is one that arises frequently in languages like Smalltalk and Lisp. [Bobrow 86] describes an approach to this sort of problem-based on a lazy copy-upon-destructive-reference-scheme.)

The **turnOnTheStimulus** and **turnOffTheStimulus** methods pass the current stimulus object to this BatteryItem's stimulus generator.

BatteryItem methods for: stimulus control

```
prepareNextStimulus
    "Tell our stimulus generator to prepare the stimulus."

    stimulusGenerator prepare: self nextTrial stimulus

prepareStimulusSequence
    "Prepare a sequence, and set up a stream over it."

    | seq stims count probs |
    stims ← self stimuli.
    count ← self trials.
    probs ← self probabilities.
    seq ← self sequenceGenerator
        generateSequenceOn: stims
        withSize: count
        andProbabilities: probs.
    self stimulusSequence: (SampledStream on: seq)

selectNextStimulus
    "Pull the next stimulus from our stimulus sequence, and tell ourself
    what it is."

    self nextTrial stimulus: self stimulusSequence next deepCopy

turnOffTheStimulus
    "Tell our stimulus generator to turn off the stimulus."

    self stimulusGenerator turnOff: self stimulus

turnOnTheStimulus
    "Tell our stimulus generator to turn on the stimulus."

    self stimulusGenerator turnOn: self stimulus
```

BatteryItem methods for: response control

The battery simulation uses discrete objects for response reporting. The response control protocol coordinates these.

The response control protocol for battery items consists of two methods. The **enableResponseReporting** method passes an **enable** method on to the response device. The **evaluateResponse** method queries the response device, and loads the fields of the returned response into the current trial object. Note that these values are stored using message sends to self, so that a different data management scheme might easily be substituted.

BatteryItem methods for: response control

```
enableResponseReporting
    "Turn on the response device."

    self responseDevice enable

evaluateResponse
    "Ask the response device for information pertaining to the current
    response, and store it."

    | resp |
    resp ← self responseDevice response.
    self responseType: resp responseButton.
    self responseTime: resp responseTime
```

BatteryItem methods for: parameter access

These methods translate queries to the battery item into queries to the BatteryItem's BatteryParameter object. Consult the Battery-Parameters category description for more information on these objects.

The use of message sends to self insulates the bulk of the code in BatteryItem from changes in the design of the data structures (which have indeed occurred during the evolution of this project).

BatteryItem methods for: parameter access

```
baseline
    ^self parameters baseline

channels
    ^self parameters channels

digitizingRate
    ^self parameters digitizingRate

inputBitA
    ^self parameters inputBitA

inputBitB
    ^self parameters inputBitB

label
    ^self parameters label

name
```

```
^self parameters name
```

outputBitA
^self parameters outputBitA

outputBitB
^self parameters outputBitB

points
^self parameters points

probabilities
^self parameters probabilities

responseMax
^self parameters responseMax

stimuli
^self parameters stimuli

stimulusDuration
^self parameters stimulusDuration

trialDuration
^self parameters trialDuration

trials
^self parameters trials

BatteryItem methods for: single trial access

These access methods translate sends to self into references to the current single trial data object. Consult the definition of class BatteryTrial in the Data-Management system category for more information.

The use of message sends to self helps to encapsulate the bulk of the battery item framework from the details of our single trial data management strategy. This strategy, like the parameter and stimulus management strategies, has evolved considerably since the start of this project.

BatteryItem methods for: single trial access

data
^self currentTrial data

data: anObject
^self currentTrial data: anObject

responseTime
^self currentTrial responseTime

responseTime: time
```
^self currentTrial responseTime: time
```

responseType
```
^self currentTrial responseType
```

responseType: type
```
^self currentTrial responseType: type
```

stimulus
```
^self currentTrial stimulus
```

stimulus: stim
```
^self currentTrial stimulus: stim
```

trialNumber
```
^self currentTrial trialNumber
```

trialNumber: n
```
^self currentTrial trialNumber: n
```

BatteryItem methods for: accessing

The methods in this category provide explicit read/write access to all of BatteryItem's instance variables.

Perhaps the most interesting thing about this protocol category is that it was generated automatically, using the **accessingSubclass: instanceVariableNames: classVariableNames: poolDictionaries: category:** method found on page 289 of the Smalltalk Blue book [Goldberg 83].

BatteryItem methods for: accessing

averagePipeline
```
^averagePipeline
```

averagePipeline: argument
```
averagePipeline ← argument.
^argument
```

block
```
^block
```

block: argument
```
block ← argument.
^argument
```

blockNumber

^blockNumber

```
blockNumber: argument
    blockNumber ← argument.
    ^argument

buffer
    ^buffer

buffer: argument
    buffer ← argument.
    ^argument

currentTrial
    ^currentTrial

currentTrial: argument
    currentTrial ← argument.
    ^argument

currentTrialNumber
    ^currentTrialNumber

currentTrialNumber: argument
    currentTrialNumber ← argument.
    ^argument

digitizer
    ^digitizer

digitizer: argument
    digitizer ← argument.
    ^argument

experimenterLog
    ^experimenterLog

experimenterLog: argument
    experimenterLog ← argument.
    ^argument

nextTrial
    ^nextTrial

nextTrial: argument
    nextTrial ← argument.
    ^argument

outputStream
    ^outputStream

outputStream: argument
    outputStream ← argument.
    ^argument
```

parameters

^parameters

parameters: argument
parameters ← argument.
^argument

responseDevice
^responseDevice

responseDevice: argument
responseDevice ← argument.
^argument

sequenceGenerator
^sequenceGenerator

sequenceGenerator: argument
sequenceGenerator ← argument.
^argument

stimulusClock
^stimulusClock

stimulusClock: argument
stimulusClock ← argument.
^argument

stimulusGenerator
^stimulusGenerator

stimulusGenerator: argument
stimulusGenerator ← argument.
^argument

stimulusSequence
^stimulusSequence

stimulusSequence: argument
stimulusSequence ← argument.
^argument

tallyPipeline
^tallyPipeline

tallyPipeline: argument
tallyPipeline ← argument.
^argument

trialClock
^trialClock

trialClock: argument
trialClock ← argument.
^argument

waveformDisplay

^waveformDisplay

```
waveformDisplay: argument
    waveformDisplay ← argument.
    ^argument
```

BatteryItem class

class BatteryItem class
superclass AccessableObject class

BatteryItem class methods for: instance creation

This BatteryItem class category overrides **new** in order to make sure that an **initialize** message is sent, and the BatteryDataDictionary is initialized. See BatteryBrowser class for an example of a use of the **defaultItemList** method.

BatteryItem class methods for: instance creation

```
defaultItemList
    "Return a default item list..."

    | list |
    list ← OrderedCollection new: 10.
    list add: SternbergTask new.
    list add: ToneOddball new.
    list add: WordOddball new.
    ^list asArray

new
    "Create a new battery item, send it an initialize message, and return
    the result..."

    | item |
    BatteryDataDictionary isNil ifTrue: [self initialize].
    item ← super new.
    item initialize.
    ^item
```

BatteryItem class methods for: instance initialization

The **initialize** method fcreates a new BatteryDataDictionary. Thehe standard fileOut format [Krasner 83] generates a <Class> initialize message send for any metaclass that implements an **initialize** method.

BatteryItem class methods for: instance initialization

initialize
BatteryDataDictionary ← DataDictionary new: 100

BatteryItem class methods for: category management

This protocol category has served as a repository for a collection of methods I have been using to manage the Smalltalk code for this project. One can make a case that from the standpoint of design aesthetics, much of this code should reside elsewhere. Nevertheless, here it is. It is presented in the hope that some of it might prove of utility to someone entrusted with maintaining a large collection of classes such the one presented here.

The **backUpEverything** method first files out all the categories in the list returned by the **batteryCategories** method as separate **<**Category-Name**>**.st files using the **fileOutCategories:** method. It then files out these same categories into a single file named 'Project-Sources.st' using the **fileOutCategories:on:** method. Then it files out the changes set [Goldberg 85] to 'Project-Changes.st' using the **fileOutChangesOn:** method.

The **fileOutBundledCategories:on:** method creates a Unix shar archive file that can create a set of individual system category *.st files on a Unix system. (Use the Bourne shell). The comment at the beginning of this method givens an example of how one might use the **matchSystemCategories:** method.

The print out messages **printOutBundledCategories:on:**, **printOutCategories** and **printOutCategories:on:** are similar in function to the analogous file out messages, except that they create Unix *troff* ready output files. These may be processed along with a macro file named *tmac.st* using the command: *rditroff -ms* **<***name***>**. (The *tmac.st* file should be in the default directory when *rditroff* is run.) The Smalltalk code to generate *troff* ready output was derived from code from the Berkeley Smalltalk distribution kit [Krasner 83].

BatteryItem class methods for: category management

backUpEverything

"Update all our back up files..." "BatteryItem backUpEverything"

```
BatteryItem fileOutCategories: BatteryItem batteryCategories.
BatteryItem
    fileOutCategories: BatteryItem batteryCategories
    on: (FileStream fileNamed: 'Project-Sources.st').
BatteryItem
    fileOutChangesOn: (FileStream fileNamed: 'Project-Changes.st')
```

batteryCategories

"Return all the battery project categories..."

```
^#('Battery-Items' 'Battery-Parameters' 'Stimulus-Generators' 'Stimulus-Support'
'Sequence-Support' 'Response-Support' 'Data-Management' 'Realtime-Support'
'Realtime-Devices' 'Waveform-Support' 'Random-Support' 'Plumbing-Support' 'Linear-
Algebra' 'Source-Hacking' 'Accessable-Objects' 'Interface-Battery' 'Interface-Protocol' )
```

fileOutBundledCategories: categories on: aFileStream

"BatteryItem fileOutBundledCategories: (BatteryItem matchSystemCategories: 'Int') on:*
(FileStream fileNamed: 'shar.Interface-Stuff.st')"

```
| file line |
categories do:
    [:category |
    Transcript cr; show: '--> ' , category.
    file ← 'troff.' , category.
    file ← file copyReplaceAll: ' ' with: '_'.
    line ← 'echo "--> "' , file.
    aFileStream nextPutAll: line; cr.
    line ← 'cat > ' , file , ' << ''!!Hehehehe...!!'''.
    aFileStream nextPutAll: line; cr.
    SystemOrganization fileOutCategory: category on: aFileStream.
    aFileStream cr; nextPutAll: '!!Hehehehe...!!'; cr].
aFileStream shorten; close
```

fileOutCategories: categories

"BatteryItem fileOutCategories: BatteryItem batteryCategories"

```
categories do:
    [:category |
    Transcript cr; show: '--> ' , category.
    SystemOrganization fileOutCategory: category]
```

fileOutCategories: categories on: aFileStream

"BatteryItem fileOutCategories: BatteryItem batteryCategories on:
(FileStream fileNamed: 'Project-Sources.st')"

```
categories do:
```

```
        [:category |
        Transcript cr; show: '--> ' , category.
        SystemOrganization fileOutCategory: category on: aFileStream.
        aFileStream cr].
    aFileStream shorten; close
```

fileOutChangesOn: aFileStream

```
    "BatteryItem fileOutChangesOn: (FileStream fileNamed: 'Project-Changes.st')"

    "Remove old DoIt code, and file out changes..."
    Smalltalk allBehaviorsDo:
        [:class | class removeSelector: #DoIt; removeSelector: #DoItIn:].
    aFileStream fileOutChanges
```

fileOutComments: categories on: aFileStream

```
    "BatteryItem fileOutComments:  BatteryItem batteryCategories on:
    (FileStream fileNamed: 'Project-Sources.comments')"

    categories do:
        [:category |
        Transcript cr; show: '--> ' , category.
        SystemOrganization fileOutComments: category on: aFileStream.
        aFileStream cr].
    aFileStream shorten; close
```

matchSystemCategories: pattern

```
    "Return a collection of category names that matches the given pattern..."

    ^SystemOrganization categories select: [:category | pattern match: category]
```

printOutBundledCategories: categories on: aFileStream

```
    "BatteryItem printOutBundledCategories: (BatteryItem matchSystemCategories:
'Int*') on:
    (FileStream fileNamed: 'shar.troff.Interface-Stuff')"

    | file line |
    categories do:
        [:category |
        file ← 'troff.' , category.
        file ← file copyReplaceAll: ' ' with: '_'.
        line ← 'echo "--> "' , file.
        aFileStream nextPutAll: line; cr.
        line ← 'cat > ' , file , ' << ''!!Hehehehe...!!'''.
        aFileStream nextPutAll: line; cr.
        Object printOutStartUp: aFileStream.
        Object printOutTimeStamp: aFileStream.
        SystemOrganization printOutCategory: category on: aFileStream.
        aFileStream cr; nextPutAll: '!!Hehehehe...!!'; cr].
    aFileStream shorten; close
```

printOutCategories: categories

```
	"Print out a list of categories..."
	"BatteryItem printOutCategories: BatteryItem batteryCategories"
	"BatteryItem printOutCategories: (BatteryItem matchSystemCategories: 'Int*')"

	categories do: [:category | SystemOrganization printOutCategory: category]
```

printOutCategories: categories on: aFileStream

```
    "BatteryItem printOutCategories: BatteryItem batteryCategories on: (FileStream
    fileNamed: 'troff.Project-Sources')"

    Object printOutStartUp: aFileStream.
    Object printOutTimeStamp: aFileStream.
    categories do:
        [:category |
        SystemOrganization printOutCategory: category on: aFileStream.
        aFileStream cr].
    aFileStream shorten; close

    "Brian Foote  9/28/86  Added start up and time stamp calls..."
```

BatteryItem initialize

SternbergTask

Class SternbergTask implements a visual Sternberg memory task [Sternberg 69] on top of the ERP/oddball framework present in BatteryItem. The variation on the Sternberg paradigm presented here involves the presentation to a subject, for each trial, of a set of characters or digits called a *memory set*. The subject is instructed to remember the characters in the memory set. After the memory set is removed from the screen, the subject is presented with another character called a *probe.* The subject's task is to indicated whether the probe belongs to the memory set. A trial in which the probe was in fact drawn from the memory set is a *positive* trial. One in which the probe was not drawn from the memory set is a*negative* trial.

class name SternbergTask
superclass BatteryItem
instance variable names
items

trialClock
stimulusClock
digitizer
responseDevice
parameters
stimulusSequence
sequenceGenerator
stimulusGenerator
buffer
experimenterLog
waveformDisplay
block
blockNumber
nextTrial
currentTrial
currentTrialNumber
outputStream
averagePipeline
tallyPipeline
class variable names
BatteryDataDictionary
category Battery-Items
hierarchy

Object
    AccessableObject
        BatteryItem
            SternbergTask
            ToneOddball
            WordOddball

SternbergTask methods for: instance initialization

The **allocateParameters** and **allocateStimulusGenerator** methods implement item specific parameter and stimulus generator resource allocation code.

SternbergTask methods for: instance initialization

```
allocateParameters
    "Allocate our parameter object."

    self parameters: SternbergTaskParameters new

allocateStimulusGenerator
    "Build our stimulus generator."

    self stimulusGenerator: SternbergDisplayGenerator new
```

SternbergTask methods for: data collection

The **collectData** method defines the experimental control and data acquisition strategy for the Sternberg task battery item. The remainder of the methods in this protocol category are inherited from BatteryItem. First, the A/D system is started. Then, the memory set is prepared, and trial clock is instructed to wait until a baseline period (relative to the start of the trial) has elapsed. The memory set is then presented for an appropriate amount of time, and turned off.

The pattern for presenting the actual stimulus (i.e. the probe) is similar. First, a probe is prepared. The program then waits until the time at which the probe is to be presented arrives. Then, a timer is started so that there will be a timebase relative to the stimulus onset. Next, the probe is turned on, and the response reporting system is enabled. When the stimulus duration has elapsed, the stimulus is turned off.

When all the data have been collected, a check is made to see whether additional time is necessary for subject response data collection.

SternbergTask methods for: data collection

```
collectData
    "Do Sternberg data collection."

"First, start the A/D converter..."
    self startDigitizer.

"Now, prepare the memory set, and present it..."
    self prepareTheMemorySet.
    self trialClock waitUntil: self memorySetBaseline.
    self turnOnTheMemorySet.
    self trialClock waitUntil: self memorySetBaseline + self memorySetDuration.
    self turnOffTheMemorySet.

"Next, start the stimulus clock and turn on the stimulus.
    Enable response reporting too.  Then wait until the stimulus
    duration has elapsed, and turn off the stimulus..."
    self prepareTheProbe.
    self trialClock waitUntil: self probeBaseline.
    self stimulusClock startFor: self trialDuration - self probeBaseline.
    self turnOnTheProbe.
    self enableResponseReporting.
    self stimulusClock waitUntil: self stimulusDuration.
    self turnOffTheProbe.

"Finally, wait for the digitizer to finish.
if the RT wait time is greater than the digitizing time,
wait that out..."
    self digitizer wait.
    self stimulusClock waitUntil: self responseMax
```

SternbergTask methods for: stimulus control

The methods in this protocol category embody a substantial proportion of the code that makes the SternbergTask battery item distinct.

The **prepareNextStimulus** method selects both a memory set and a probe for the next trial. The way in which the memory set is selected shows a novel application of SampledStreams. The memory set should be a set of a given size drawn at random (without replacement) from a given vocabulary of characters.

This set is constructed as follows: First, a SampledStream is created over the BatteryItem's vocabulary string. Then an empty output string (the memory set) is created, and a WriteStream is defined over it (memorySetStream). The memory set is constructed by moving the appropriate number of characters from the input (sampled) stream to the output stream. The underlying collection of this output stream is the memory set, which can then be used.

The probe is selected and prepared as follows: If a positive trial is called for, a single character is selected at random, using a RandomStream (sample with replacement) over the memory set. If a negative probe is required, the next character in the vocabulary stream that was used to create the memory set, from which the memory set has already been drawn, is selected as the stimulus character.

The **prepareTheMemorySet** method simply forwards the current stimulus descriptor to the stimulus generator. The remaining methods in this protocol category similarly forward stimulus manipulation requests to the stimulus generator.

SternbergTask methods for: stimulus control

```
prepareNextStimulus
    "Select a memory set and a probe."

    |vocabularyStream memorySet memorySetStream |
    vocabularyStream ← SampledStream on: self vocabulary.
    memorySet ← String new: self memorySetSize.
    memorySetStream ← WriteStream on: memorySet.
    self memorySetSize timesRepeat: [memorySetStream nextPut: vocabularyStream
next].
    self nextTrial stimulus memorySet: memorySet.
    self nextTrial stimulus category = #positive
        ifTrue: [ self nextTrial stimulus probe:
            (RandomStream on: memorySet) next asSymbol asString ]
        ifFalse: [self nextTrial stimulus probe:
            vocabularyStream next asSymbol asString]

prepareTheMemorySet
    "Tell our stimulus generator to get the memory set ready."

    self stimulusGenerator prepareMemorySet: self currentTrial stimulus

prepareTheProbe
    "Tell our stimulus generator to get the memory set ready."

    self stimulusGenerator prepareProbe: self currentTrial stimulus

turnOffTheMemorySet
    "Tell our stimulus generator to turn off the memory set."
```

```
self stimulusGenerator turnOffMemorySet: self currentTrial stimulus
```

turnOffTheProbe

"Tell our stimulus generator to turn off the probe."

```
self stimulusGenerator turnOffProbe: self currentTrial stimulus
```

turnOnTheMemorySet

"Tell our stimulus generator to turn on the memory set."

```
self stimulusGenerator turnOnMemorySet: self currentTrial stimulus
```

turnOnTheProbe

"Tell our stimulus generator to turn on the probe."

```
self stimulusGenerator turnOnProbe: self currentTrial stimulus
```

SternbergTask methods for: parameter access

The methods in this category add BatteryItem access methods for the fields that the SternbergTaskParameters object adds to the BatteryParameters object.

SternbergTask methods for: parameter access

memorySetBaseline

```
^self parameters memorySetBaseline
```

memorySetDuration

```
^self parameters memorySetDuration
```

memorySetSize

```
^self parameters memorySetSize
```

probeBaseline

```
^self parameters probeBaseline
```

probeDuration

```
^self parameters probeDuration
```

vocabulary

```
^self parameters vocabulary
```

SternbergTask methods for: data processing

Each concrete battery item provides its own **prepareAverageFixtures** method. This is necessary to take into account item specific stimulus characteristics and labeling requirements. A detailed discussion of the mechanics of fixture preparation can be found in the discussion of this protocol category given for class BatteryItem.

SternbergTask methods for: data processing

prepareAverageFixtures

```
"Prepare the average pipelines."

self averagePipeline: (Dictionary new: 10).
self block averages: (Dictionary new: 10).
self
    average: [self currentTrial data]
    using: [:data | true]
    withLabel: ’Average (All trials)’.
self
    average: [self currentTrial data]
    using: [:data | self currentTrial stimulus category = #positive]
    withLabel: ’Average (Positive)’.
self
    average: [self currentTrial data]
    using: [:data | self currentTrial stimulus category = #negative]
    withLabel: ’Average (Negative)’
```

SternbergTask class

class	SternbergTask class
superclass	BatteryItem class

SternbergTask class methods for: examples

The **example1** method shows how one might execute a sample block of trials. Note that before a block can be run, the Timebase object must have been initialized. In practice, battery blocks will usually be run by pressing the **run** button after having selected an item in the battery browser.

SternbergTask class methods for: examples

example1

```
"SternbergTask example1"

Timebase initialize.
SternbergTask new doBlock
```

ToneOddball

The ToneOddball class implements a simple auditory oddball experiment [Duncan-Johnson 77]. In this experiment, a subject is presented with one of two tones. The subject can be instructed to count one of the tones, and ignore the other. The subject can also be told to press a button in response to one or both stimulus categories. A frequent manipulation is to vary the relative frequency of the two stimulus categories.

class name ToneOddball
superclass BatteryItem
instance variable names
items

trialClock
stimulusClock
digitizer
responseDevice
parameters
stimulusSequence
sequenceGenerator
stimulusGenerator
buffer
experimenterLog
waveformDisplay
block
blockNumber
nextTrial
currentTrial
currentTrialNumber
outputStream
averagePipeline
tallyPipeline
class variable names
BatteryDataDictionary
pool dictionaries
category Battery-Items
hierarchy

```
Object
    AccessableObject
        BatteryItem
            SternbergTask
            ToneOddball
            WordOddball
```

ToneOddball methods for: instance initialization

The **allocateParameters** and **allocateStimulusGenerator** methods implement item specific parameter and stimulus generator resource allocation code.

ToneOddball methods for: instance initialization

allocateParameters

```
"Allocate our parameter object."

self parameters: ToneOddballParameters new
```

allocateStimulusGenerator

```
"Build our stimulus generator."

| bitA bitB |
bitA ← self outputBitA.
bitB ← self outputBitB.
self stimulusGenerator: (ToneGenerator bitA: bitA bitB: bitB)
```

ToneOddball methods for: parameter access

ToneOddball methods for: parameter access

outputBitA

```
^self parameters outputBitA
```

outputBitB

```
^self parameters outputBitB
```

ToneOddball methods for: data collection

The **collectData** method specifies the experimental control and data acquisition scheme used for this item. First, the digitizer is started. Then the method waits for a baseline period, relative to the start of this trial, to elapse. Next, the stimulus timer is started. Immediately thereafter, the stimulus is turned on and response reporting is enabled. When the stimulus presentation period has elapsed, the stimulus is turned off. The method then waits until data collection is complete, and the response window time has elapsed.

ToneOddball methods for: data collection

```
collectData
    "Do tone oddball data collection."

"First, start the A/D converter, and wait until the baseline elapses..."
    self startDigitizer.
    self trialClock waitUntil: self baseline.

"Next, start the stimulus clock and turn on the stimulus.
Enable response reporting too.  Then wait until the stimulus
duration has elapsed, and turn off the stimulus..."
    self stimulusClock startFor: self trialDuration - self baseline.
    self turnOnTheStimulus.
    self enableResponseReporting.
    self stimulusClock waitUntil: self stimulusDuration.
    self turnOffTheStimulus.

"Finally, wait for the digitizer to finish.
If the RT wait time is greater than the digitizing time,
wait that out..."
    self digitizer wait.
    self stimulusClock waitUntil: self responseMax
```

ToneOddball methods for: data processing

Each concrete battery item provides its own **prepareAverageFixtures** method. This is necessary to take into account item specific stimulus characteristics and labeling requirements. A detailed discussion of the mechanics of fixture preparation can be found in the discussion of this protocol category given for class BatteryItem.

ToneOddball methods for: data processing

```
prepareAverageFixtures
    "Prepare the average pipelines."

    self averagePipeline: (Dictionary new: 10).
    self block averages: (Dictionary new: 10).
    self
        average: [self currentTrial data]
        using: [:data | true]
        withLabel: 'Average (All trials)'.
    self
        average: [self currentTrial data]
        using: [:data | self currentTrial stimulus category = #catA]
        withLabel: 'Average (Category A)'.
    self
```

```
average: [self currentTrial data]
using: [:data | self currentTrial stimulus category = #catB]
withLabel: 'Average (Category B)'
```

ToneOddball class

class	ToneOddball class
superclass	BatteryItem class

ToneOddball class methods for: examples

The **example1** method shows how one might execute a sample block of trials. Note that before a block can be run, the Timebase object must have been initialized. In practice, battery blocks will usually be run by pressing the **run** button after having selected an item in the battery browser.

ToneOddball class methods for: examples

example1

```
"ToneOddball example1"

Timebase initialize.
ToneOddball new doBlock
```

WordOddball

The WordOddball class implements a visual semantic oddball task [Donchin 81]. In this experiment, a subject is presented with a random word drawn from one of two categories. The subject must count or respond as with the ToneOddball.

It is instructive to point out that to implement a concrete BatteryItem, it is necessary for an experiment to implement just four methods: **allocateParameters**, **allocateStimulusGenerator**, **collectData**, and **prepareAverageFixtures.** To fully exploit the battery framework, an experimenter might also need to create new subclasses of BatteryParameter, StimulusGenerator, and Stimulus hierarchies.

class name WordOddball
superclass BatteryItem
instance variable names
items

trialClock
stimulusClock
digitizer
responseDevice
parameters
stimulusSequence
sequenceGenerator
stimulusGenerator
buffer
experimenterLog
waveformDisplay
block
blockNumber
nextTrial
currentTrial
currentTrialNumber
outputStream
averagePipeline
tallyPipeline
class variable names
BatteryDataDictionary
category Battery-Items
hierarchy

```
Object
    AccessableObject
        BatteryItem
            SternbergTask
            ToneOddball
```

WordOddball methods for: instance initialization

Subclasses of BatteryItem are required to provide implementations of two standard resource allocation messages: **allocateParameters**, and **allocateStimulusGenerator**. (Both are defined as being subclass responsibilities in BatteryItem.) The **allocateParameters** method simply ensures that a parameter object belonging to the correct class is allocated. The **allocateStimulusGenerator** message's task is potentially more complicated. It must ensure that an instance of some sort of StimulusGenerator appropriate to the type of battery item being constructed is created and initialized.

The battery simulation has attempted to separate battery items and (simulated) stimulus generation equipment so that stimulus generators might be mixed and matched among several different battery items. Frequently, the major differences among a collection of experimental designs will be in sorts of stimuli they employ. By separating the stimulus generation code from the rest of the experimental code, the reusability of both should be enhanced.

The stimulus generation and response reporting hierarchies are examples of places in this battery simulation where tasks handled in the original CPL battery by (simulated) inheritance are now handled by discrete components. The evolution of inheritance frameworks into component frameworks is discussed in Chapter VI.

The **allocateParameters** and **allocateStimulusGenerator** methods implement item specific parameter and stimulus generator resource allocation code. Of interest here is the scheme we use in **allocateStimulusGenerator** for implementing our random word lists. A dictionary of streams is created which is keyed by the two stimulus categories. Each stream is a RandomStream over a vocabulary array. Since RandomStreams sample with replacement, they will provide, each time they are asked, a random word from their respective underlying vocabularies. This dictionary, once constructed, is given to a newly created WordGenerator (a subclass of StimulusGenerator).

WordOddball methods for: instance initialization

```
allocateParameters
    "Allocate our parameter object."

    self parameters: WordOddballParameters new


allocateStimulusGenerator
    "Build our stimulus generator."

    | streamDictionary animals vegetables |
    streamDictionary ← Dictionary new: 2.
    animals ← #(dog cat mouse ).
    vegetables ← #(carrot potato tomato ).
    streamDictionary at: #animal put: (RandomStream on: animals).
    streamDictionary at: #vegetable put: (RandomStream on: vegetables).
    self stimulusGenerator: (WordGenerator on: streamDictionary)
```

WordOddball methods for: data collection

Each subclass of BatteryItem is required to implement a **collectData** method. This method conducts the basic per-trial data collection and experimental control activity for each battery item.

The **collectData** method specifies experimental control and data acquisition scheme used for this item. First, the digitizer is started. Then the method waits for a baseline period, relative to the start of this trial, to elapse. Then, the stimulus timer is started. Immediately thereafter, the method turns on the stimulus and enable response reporting. When the stimulus presentation period has elapsed, the stimulus is turned off. The method then waits until data collection is complete, and the response window time has elapsed.

The perceptive reader will note that the **collectData** method given here is identical to the one given in ToneOddball. I could have either defined an intermediate abstract calls (say, SimpleOddball) between BatteryItem and these two classes to take advantage of this. Alternately, I could have made this implementation of **collectData** the default in BatteryItem itself. That I did neither is in recognition of the fact that in a somewhat fuller battery simulation, detail that would distinguish a larger set of applications from one another would quickly emerge at this level, just as it does in the SternbergTask battery item.

WordOddball methods for: data collection

```
collectData
    "Do word oddball data collection."

"First, start the A/D converter, and wait until the baseline elapses..."
    self startDigitizer.
    self trialClock waitUntil: self baseline.

"Next, start the stimulus clock and turn on the stimulus.
Enable response reporting too.  Then wait until the stimulus
duration has elapsed, and turn off the stimulus..."
    self stimulusClock startFor: self trialDuration - self baseline.
    self turnOnTheStimulus.
    self enableResponseReporting.
    self stimulusClock waitUntil: self stimulusDuration.
    self turnOffTheStimulus.

"Finally, wait for the digitizer to finish.
If the RT wait time is greater than the digitizing time,
wait that out..."
    self digitizer wait.
    self stimulusClock waitUntil: self responseMax
```

WordOddball methods for: data processing

Each concrete battery item must provide its own **prepareAverageFixtures** method. This is necessary to take into account item specific stimulus characteristics and labeling requirements. A detailed discussion of the mechanics of fixture preparation can be found in the discussion of this protocol category given for class BatteryItem.

WordOddball methods for: data processing

```
prepareAverageFixtures
    "Prepare the average pipelines."

    self averagePipeline: (Dictionary new: 10).
    self block averages: (Dictionary new: 10).
    self
        average: [self currentTrial data]
        using: [:data | true]
        withLabel: 'Average (All trials)'.
    self
        average: [self currentTrial data]
        using: [:data | self currentTrial stimulus category = #catA]
        withLabel: 'Average (Category A)'.
```

```
self
    average: [self currentTrial data]
    using: [:data | self currentTrial stimulus category = #catB]
    withLabel: 'Average (Category B)'
```

WordOddball class

class WordOddball class
superclass BatteryItem class

WordOddball class methods for: examples

The **example1** method shows how one might execute a sample block of trials. Note that before a block can be run, the Timebase object must have been initialized. In practice, battery blocks will usually be run by pressing the **run** button after having selected an item in the battery browser.

WordOddball class methods for: examples

```
example1
    "WordOddball example1"

    Timebase initialize.
    WordOddball new doBlock
```

Battery-Parameters

The Battery-Parameters system category contains the following classes:

BatteryParameters	Houses common BatteryItem parameters
SternbergTaskParameters	Adds SternbergTask specific parameters
ToneOddballParameters	Adds ToneOddball specific parameters
WordOddballParameters	Adds WordOddball specific parameters

These parameter objects are in effect records or dictionaries that contain those values that can be edited and modified by the experimenter. (In fact, since these objects are subclasses of AccessableObject, they can be treated as records or dictionaries.)

Parameter objects are distinct objects rather than parts of the items themselves for several reasons. One is that, as separate objects, they can be copied and archived along with the collected data without recording the entire state of the item itself. This distinction between the parameter objects and the item objects could become more significant should an external data management scheme be implemented. The second reason for keeping parameter objects separate is to distinguish values that should be presented to the user for alteration via the battery user interface from the battery item's other instance variables. Parameter objects might also serve as a components in a more general (and yet to be designed) user parameter management/interface scheme.

BatteryParameters

class name BatteryParameters
superclass AccessableObject
instance variable names

items

baseline
channels
points
digitizingRate
responseMax
stimulusDuration
trialDuration
trials
writeDataFlag
probabilities
stimuli
inputBitA
inputBitB
name
label

category Battery-Parameters
hierarchy

```
Object
    AccessableObject
        BatteryParameters
            SternbergTaskParameters
            ToneOddballParameters
            WordOddballParameters
```

BatteryParameters methods for: instance initialization

These two methods establish default values for each basic parameter. Note that by default data are written to the data dictionary, and that the probability and stimuli fields are Arrays.

BatteryParameters methods for: instance initialization

```
establishDefaults
    "Set out our defaults..."

    baseline ← 1.
    channels ←  1.
    points ←  10.
    digitizingRate ←  1.
    responseMax ←  9.
    stimulusDuration ←  2.
```

```
trialDuration ← 15.
trials ← 10.
writeDataFlag ←  True.
probabilities ← #(0.5 0.5).
stimuli ← #(catA catB)
```

initialize
"Set defaults and return ourself..."

```
self establishDefaults.
^self
```

BatteryParameters methods for: accessing

These methods allow explicit read/write access to all of this object's fields using the conventional record style protocol. (Note that since we are a subclass of AccessableObject, all these messages would have functioned correctly anyway, albeit much less quickly.)

BatteryParameters methods for: accessing

baseline
```
^baseline
```

baseline: argument
```
baseline ← argument.
^argument
```

channels
```
^channels
```

channels: argument
```
channels ← argument.
^argument
```

digitizingRate
```
^digitizingRate
```

digitizingRate: argument
```
digitizingRate ← argument.
^argument
```

inputBitA
```
^inputBitA
```

inputBitA: argument
```
inputBitA ← argument.
^argument
```

inputBitB
```
^inputBitB
```

inputBitB: argument
```
inputBitB ← argument.
^argument
```

label
^label

label: argument
label ← argument.
^argument

name
^name

name: argument
name ← argument.
^argument

points
^points

points: argument
points ← argument.
^argument

probabilities
^probabilities

probabilities: argument
probabilities ← argument.
^argument

responseMax
^responseMax

responseMax: argument
responseMax ← argument.
^argument

stimuli
^stimuli

stimuli: argument
stimuli ← argument.
^argument

stimulusDuration
^stimulusDuration

stimulusDuration: argument
stimulusDuration ← argument.
^argument

trialDuration
^trialDuration

trialDuration: argument

```
trialDuration ← argument.
^argument
```

trials

```
^trials
```

```
trials: argument
    trials ← argument.
    ^argument

writeDataFlag
    ^writeDataFlag

writeDataFlag: argument
    writeDataFlag ← argument.
    ^argument
```

BatteryParameter class

class	BatteryParameter class
superclass	AccessableObject class

BatteryParameters class methods for: instance creation

The **new** method is overridden here to enforce the convention that each new object must explicitly call an **initialize** method.

BatteryParameters class methods for: instance creation

```
new
    "Create a new parameter object and initialize it..."

    ^super new initialize
```

SternbergTaskParameters

class name SternbergTaskParameters
superclass BatteryParameters
instance variable names

items

baseline
channels
points
digitizingRate
responseMax
stimulusDuration
trialDuration
trials
writeDataFlag
probabilities
stimuli
inputBitA
inputBitB
name
label

memorySetBaseline
memorySetDuration
probeBaseline
probeDuration
memorySetSize
vocabulary

class variable names
pool dictionaries
category Battery-Parameters
hierarchy

Object
    AccessableObject
        BatteryParameters
            SternbergTaskParameters
            ToneOddballParameters
            WordOddballParameters

SternbergTaskParameters methods for: instance initialization

The **establishDefaults** method overrides the one given in BatteryParameters so that values can be established for item specific parameters. Note that its first official act is to initialize the common parameters via a call to its superclass.

SternbergTaskParameters methods for: instance initialization

```
establishDefaults
    "Do Sternberg defaults... "

    | positive negative |
    super establishDefaults.
    memorySetBaseline ← 1.
    memorySetDuration ← 1.
    memorySetSize ← 3.
    vocabulary ← 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
    probeBaseline ← 4.
    probeDuration ← 1.
    positive ← SternbergStimulus new.
    positive category: #positive.
    negative ← SternbergStimulus new.
    negative category: #negative.
    stimuli ← Array with: positive with: negative.
    name ← 'Sternberg/ERP Task'.
    label ← 'Sternberg/ERP Experiment'
```

SternbergTaskParameters methods for: accessing

Allow explicit read/write access to all of this object's fields using the conventional record style protocol. (Note that since SternbergTaskParameters is a subclass of AccessableObject, all these messages would have functioned correctly anyway, albeit much less quickly.)

SternbergTaskParameters methods for: accessing

```
memorySetBaseline
    ^memorySetBaseline

memorySetBaseline: argument
    memorySetBaseline ← argument.
    ^argument

memorySetDuration
    ^memorySetDuration

memorySetDuration: argument
    memorySetDuration ← argument.
    ^argument

memorySetSize
    ^memorySetSize

memorySetSize: argument
```

```
memorySetSize ← argument.
^argument
```

probeBaseline
    ^probeBaseline

probeBaseline: argument
    probeBaseline ← argument.
    ^argument

probeDuration
    ^probeDuration

probeDuration: argument
    probeDuration ← argument.
    ^argument

vocabulary
    ^vocabulary

vocabulary: argument
    vocabulary ← argument.
    ^argument

ToneOddballParameters

class name ToneOddballParameters
superclass BatteryParameters
instance variable names

items

baseline
channels
points
digitizingRate
responseMax
stimulusDuration
trialDuration
trials
writeDataFlag
probabilities
stimuli
inputBitA
inputBitB
name
label

outputBitA
outputBitB

class variable names
pool dictionaries
category Battery-Parameters
hierarchy

Object
    AccessableObject
        BatteryParameters
            SternbergTaskParameters
            ToneOddballParameters
            WordOddballParameters

ToneOddballParameters methods for: instance initialization

The **establishDefaults** method overrides the one given in BatteryParameters so that values can be established for of this object's item specific parameters. Note that its first official act is to initialize the common parameters via a call to its superclass.

ToneOddballParameters methods for: instance initialization

```
establishDefaults
    "Do ToneOddball defaults... "

    | toneA toneB |
    super establishDefaults.
    toneA ← ToneStimulus new.
    toneA primary: #toneA ; category: #catA.
    toneB ← ToneStimulus new.
    toneB primary: #toneB ; category: #catB.
    stimuli ← Array with: toneA with: toneB.
    outputBitA ← 0.
    outputBitB ← 1.
    name ← 'Tone Oddball'.
    label ←  'Auditory Oddball Experiment'
```

ToneOddballParameters methods for: accessing

Allow explicit read/write access to all of this object's fields using the conventional record style protocol. (Note that since ToneOddballParameters is a subclass of AccessableObject, all these messages would have functioned correctly anyway, albeit much less quickly.)

ToneOddballParameters methods for: accessing

```
outputBitA
    ^outputBitA

outputBitA: argument
    outputBitA ← argument.
    ^argument

outputBitB
    ^outputBitB

outputBitB: argument
    outputBitB ← argument.
    ^argument
```

WordOddballParameters

class name WordOddballParameters
superclass BatteryParameters
instance variable names

items

baseline
channels
points
digitizingRate
responseMax
stimulusDuration
trialDuration
trials
writeDataFlag
probabilities
stimuli
inputBitA
inputBitB
name
label

class variable names
pool dictionaries
category Battery-Parameters
hierarchy

```
Object
    AccessableObject
        BatteryParameters
            SternbergTaskParameters
            ToneOddballParameters
            WordOddballParameters
```

WordOddballParameters methods for: instance initialization

The **establishDefaults** method overrides the one given in BatteryParameters so that values can be established for this object's item specific parameters.

WordOddballParameters methods for: instance initialization

```
establishDefaults
    "Do WordOddball defaults... "

    | animal vegetable |
    super establishDefaults.
    animal ← WordStimulus new.
    animal primary: #animal ; category: #catA.
    vegetable ← WordStimulus new.
    vegetable primary: #vegetable ; category: #catB.
```

stimuli ← Array with: animal with: vegetable.
name ← ’Word Oddball’.
label ← ’Animal/vegetable Oddball Experiment’

Stimulus-Generators

These classes simulate the hardware and software one would normally use to generate experimental stimuli. These classes might provide a basis for a mix-and-match stimulus generation capability in a fuller battery simulation.

Stimulus generator objects are designed to be self-contained embodiments of specific stimulus generation schemes. Since these schemes themselves vary with experimental designs as well as with the basic hardware and software characteristics of the stimulus generators, the external protocols for these objects vary as well. Note that though the stimulus generator classes defined here exactly mirror the BatteryItem hierarchy, this might not necessarily be the case in a fuller simulation.

The StimulusGenerator hierarchy shows how a portion of the battery simulation that was formerly embedded in the BatteryItem hierarchy has, under Smalltalk, evolved into a separate part of the overall battery framework. The reuse of pluggable, black-box components is, all other things being equal, preferable to reuse via inheritance. The identification of components that may be factored out of an inheritance hierarchy in this fashion is usually possible only after several iterations through the design process. (This issue is discussed in detail in Chapter VI.)

Indeed, the StimulusGenerator hierarchy is in some respects still in transition between membership in the original battery hierarchy and clean isolation from it. The degree of coupling seen in the interface between each battery item and these classes is still somewhat high.

One very intriguing route by which this coupling might be reduced is via the introduction of parallelism. Under such an approach, stimulus generators would run in parallel with battery items, with both tied to the same underlying timebase. Such an approach would simplify both the battery item stimulus coordination code, and the stimulus sequencing code in this stimulus generators. Hence, the use of parallelism could greatly increase the generality and reusability of components of both hierarchies.

Realtime application programs frequently exhibit a structure in which the mainline code must explicitly schedule sequences of otherwise unrelated events. The use of parallel processes can free the designer from this temporal yoke.

The classes in this category are:

StimulusGenerator	Abstract StimulusGenerator superclass
SternbergDisplayGenerator	The SternbergTask stimulus generator
ToneGenerator	The ToneOddball stimulus generator
WordGenerator	The WordOddball stimulus generator

StimulusGenerator

The StimulusGenerator class is an abstract superclass for a family of stimulus generation objects. It defines a common, default protocol for stimulus generators.

class name StimulusGenerator
superclass Object
instance variable names
class variable names
pool dictionaries
category Stimulus-Generators
hierarchy

```
Object
    StimulusGenerator
        SternbergDisplayGenerator
        ToneGenerator
        WordGenerator
```

StimulusGenerator methods for: instance initialization

StimulusGenerator methods for: instance initialization

initialize
"Provide nothing as our default behavior..."

^self

StimulusGenerator methods for: stimulus control

This protocol defines a default stimulus control protocol to which most clients of any subclass of StimulusGenerator will subscribe. Note that not all subclasses of StimulusGenerator will override all these methods. In these cases, the default action (given below) will be to do nothing. Some subclasses of StimulusGenerator will need to provide a superset of the protocol given below for stimulus control. Such classes will add additional methods to this protocol.

StimulusGenerator methods for: stimulus control

finishBlock
"Default shall be to do nothing..."

prepare: aStimulus
"Default shall be to do nothing..."

startBlock

"Default shall be to do nothing..."

turnOff: aStimulus

"Default shall be to do nothing..."

turnOn: aStimulus

"Default shall be to do nothing..."

StimulusGenerator class

class	StimulusGenerator class
superclass	Object class

StimulusGenerator class methods for: instance creation

StimulusGenerator class methods for: instance creation

new

"Return an initialized instance..."

^super new initialize

SternbergDisplayGenerator

The SternbergDisplayGenerator class simulates the hardware and software that one would employ to generate the stimuli for a Sternberg task.

class name SternbergDisplayGenerator
superclass StimulusGenerator
instance variable names
class variable names
pool dictionaries
category Stimulus-Generators
hierarchy

```
Object
    StimulusGenerator
        SternbergDisplayGenerator
        ToneGenerator
        WordGenerator
```

SternbergDisplayGenerator methods for: stimulus control

Since the stimulus control scheme for a Sternberg task is different from the generic oddball scheme outlined in StimulusGenerator, we override the default (do nothing) behaviors for the default StimulusGenerator protocol using **shouldNotImplement.** The display device itself is simulated by writing the stimulus to the Smalltalk system transcript. Other messages in the stimulus generation protocol simply do nothing what so ever in this simulation.

This class is probably misplaced in the StimulusGenerator hierarchy. The fact that it in effect removes nearly the entire StimulusGenerator protocol is evidence that this hierarchy needs reorganization. This class is an example of a stimulus generation scheme that uses multiple stimulus events. In the CPL Battery, two items exhibited this structure. The other items exhibited the simpler structure that is reflected in the standard protocol for stimulus generation given in StimulusGenerator.

I deferred a major reorganization of this hierarchy in recognition of the potential a process-based organization might bring to the battery framework.

SternbergDisplayGenerator methods for: stimulus control

prepare: aStimulus
"We distinguish between probes and memory sets, so we don't do this..."

self shouldNotImplement

prepareMemorySet: aStimulus
"Do nothing here for now..."

prepareProbe: aStimulus
"Do nothing here for now..."

turnOff: aStimulus
"We distinguish between probes and memory sets, so we don't do this..."

self shouldNotImplement

turnOffMemorySet: aStimulus
"Do nothing here for now..."

turnOffProbe: aStimulus
"Do nothing here for now..."

turnOn: aStimulus
"We distinguish between probes and memory sets, so we don't do this..."

self shouldNotImplement

turnOnMemorySet: aStimulus
"Just write the memory set to the transcript for now..."

Transcript show: '--> Memory Set: ' , aStimulus memorySet; cr; endEntry

turnOnProbe: aStimulus
"Just write the probe to the transcript for now..."

Transcript show: '--> Probe: ' , aStimulus probe; cr; endEntry

ToneGenerator

The ToneGenerator class simulates a digital output driven two channel tone generator. Tones are generated by sending **set** and **clear** messages to instances of OutputBit.

class name ToneGenerator
superclass StimulusGenerator
instance variable names
    bitA
    bitB
class variable names
pool dictionaries
category Stimulus-Generators
hierarchy
    Object
        StimulusGenerator
            SternbergDisplayGenerator
            ToneGenerator
            WordGenerator

ToneGenerator methods for: accessing

These methods provide read/write access to the instance variables that house the instances of OutputBit that activate the (simulated) tone generators.

ToneGenerator methods for: accessing

bitA
"Return the output bit for the first tone..."

^bitA

bitA: aBit
"Set the output bit for the first tone..."

^bitA ← aBit

bitB
"Return the output bit for the second tone..."

^bitB

bitB: aBit
"Set the output bit for the second tone..."

^bitB ← aBit

ToneGenerator methods for: stimulus control

The methods in this category simulate a digital output bit driven tone generation scheme.

When a block is begun, the **startBlock** method sends **clear** messages to both the output bits to ensure that they are off. The same thing occurs at the end of a block in **finishBlock**.

The **turnOn:** and **turnOff:** methods both accept a stimulus descriptor, and check its *primary* stimulus field. If this field is #toneA, the output bit A is affected, and if it is #toneB, output bit B is affected.

ToneGenerator methods for: stimulus control

```
finishBlock
    "Make sure both our bits are off..."

    bitA clear.
    bitB clear

startBlock
    "Make sure both our bits are off..."

    bitA clear.
    bitB clear

turnOff: aStimulus
    "Turn off the bit indicated as the primary stimulus by the Stimulus object
    that was passed to us..."

    | primary |
    primary ← aStimulus primary.
    primary == #toneA ifTrue: [bitA clear. ^self].
    primary == #toneB ifTrue: [bitB clear. ^self].
    self error: 'Bad stimulus...'

turnOn: aStimulus
    "Turn on the bit indicated as the primary stimulus by the Stimulus object
    that was passed to us..."

    | primary |
    primary ← aStimulus primary.
    primary == #toneA ifTrue: [bitA set. ^self].
    primary == #toneB ifTrue: [bitB set. ^self].
```

```
        self error: 'Bad stimulus...'
```

ToneGenerator class

class ToneGenerator class
superclass StimulusGenerator class

ToneGenerator class methods for: instance creation

The **bitA:bitB:** message provides a shorthand for creating a ToneGenerator and assigning it a pair of output bits. (Perhaps **new** should be overridden to make this form of initialization mandatory.)

ToneGenerator class methods for: instance creation

```
bitA: a bitB: b
    "Create a tone generator, and set up the given bits..."

    | gen |
    gen ← super new.
    gen initialize.
    gen bitA: (OutputBit onBit: a).
    gen bitB: (OutputBit onBit: b).
    ^gen
```

WordGenerator

The WordGenerator class simulates the hardware and software one might employ for presenting word stimuli in a word oddball task. It is provided, upon creation, with a dictionary of stream objects that provide it with the word stimuli themselves. This dictionary is keyed using the *primary* field of the stimulus descriptor.

class name WordGenerator
superclass StimulusGenerator
instance variable names
streamDictionary
class variable names
pool dictionaries
category Stimulus-Generators
hierarchy

```
Object
    StimulusGenerator
        SternbergDisplayGenerator
        ToneGenerator
        WordGenerator
```

WordGenerator methods for: stimulus control

The **prepare:** method reads the next word from the stimulus stream for the type of stimulus given by the *primary* field of the stimulus descriptor and stores it in the *word* field.

The **turnOn:** method simulates the display of a word stimulus by writing it to the system transcript.

WordGenerator methods for: stimulus control

```
prepare: aStimulus
    "Load a word into the stimulus object..."

    | word stream |
    stream ← streamDictionary at: aStimulus primary.
    word ← stream next.
    aStimulus word: word

turnOn: aStimulus
    "For now, send the word to the transcript..."
```

```
Transcript show: aStimulus word ; cr ; endEntry
```

WordGenerator methods for: accessing

WordGenerator methods for: accessing

```
streamDictionary
    ^streamDictionary

streamDictionary: argument
    streamDictionary ← argument.
    ^argument
```

WordGenerator class

class	WordGenerator class
superclass	StimulusGenerator class

WordGenerator class methods for: instance creation

The **on:** creation message allows a word stream dictionary to be specified for a WordGenerator as it is created. This dictionary contains one entry for each primary stimulus type the client defines. These entries are keyed by this stimulus type, and should address a stream which will provide successive words for that stimulus category.

Note that these streams might be RandomStreams over a relatively small vocabulary (as in the implementation of WordOddball given herein), or actual word lists. The stream dictionary scheme isolates this class from such detail.

WordGenerator class methods for: instance creation

```
on: aStreamDictionary
    "Create a WordGenerator, and tell it to remember the stream dictionary we
    were given..."

    | generator |
    generator ← super new.
    generator initialize.
    generator streamDictionary: aStreamDictionary.
    ^generator
```

Stimulus-Support

This system category defines a family of stimulus descriptor objects which are manipulated by the various objects concerned with stimulus identity, such as the battery items themselves, the stimulus sequencing and generation objects, the data management system, and the averaging and statistics collection code. The objects in this category are:

Stimulus	Defines the basic stimulus descriptor
SternbergStimulus	Adds Sternberg specific stimulus fields
ToneStimulus	Adds ToneOddball specific fields
WordStimulus	Adds WordOddball specific fields

Stimulus

Stimulus is an abstract superclass for a family of stimulus objects. It defines a field for a stimulus category indicator of some sort.

class name Stimulus
superclass Object
instance variable names
category
class variable names
pool dictionaries
category Stimulus-Support
hierarchy

```
Object
    Stimulus
        SternbergStimulus
        ToneStimulus
        WordStimulus
```

Stimulus methods for: accessing

Stimulus methods for: accessing

category

```
"Return the stimulus category..."

^ category
```

category: aCategory

```
"Set the stimulus category..."

^ category ← aCategory
```

SternbergStimulus

The SternbergStimulus class adds fields for the memory set and probe stimulus to Stimulus.

class name SternbergStimulus
superclass Stimulus
instance variable names
category

memorySet
probe
class variable names
pool dictionaries
category Stimulus-Support
hierarchy
Object
Stimulus
SternbergStimulus
ToneStimulus
WordStimulus

SternbergStimulus methods for: accessing

SternbergStimulus methods for: accessing

```
memorySet
    ^ memorySet

memorySet: argument
    memorySet ← argument.
    ^ argument

probe
    ^ probe

probe: argument
    probe ← argument.
    ^ argument
```

ToneStimulus

The ToneStimulus class adds a field for the primary stimulus type to Stimulus.

class name ToneStimulus
superclass Stimulus
instance variable names
category

primary
class variable names
pool dictionaries
category Stimulus-Support
hierarchy
Object
Stimulus
SternbergStimulus
ToneStimulus
WordStimulus

ToneStimulus methods for: accessing

ToneStimulus methods for: accessing

```
primary
    ^ primary

primary: argument
    primary ← argument.
    ^ argument
```

WordStimulus

The ToneStimulus class adds a field for the primary stimulus type to Stimulus.

class name WordStimulus
superclass Stimulus
instance variable names
category

primary
word
class variable names
pool dictionaries
category Stimulus-Support
hierarchy
Object
Stimulus
SternbergStimulus
ToneStimulus
WordStimulus

WordStimulus methods for: accessing

WordStimulus methods for: accessing

primary
^ primary

primary: argument
primary ← argument.
^ argument

word
^ word

word: argument
word ← argument.
^ argument

Sequence-Support

The Sequence-Support system category contains two classes which implement components of the Battery stimulus sequence generation scheme. These classes are:

WeightedCollection	A collection with fixed proportions
SequenceGenerator	A stimulus sequence constructor

WeightedCollection

A WeightedCollection is a collection of objects in which each object in a given base set is constrained to appear with a given relative frequency. For example, a WeightedCollection over #(dog cat mouse) of size nine would result in a collection in which each of dog, cat, and mouse were present three times.

class name WeightedCollection
superclass RunArray
instance variable names
runs
values

weights
class variable names
category Sequence-Support
hierarchy

```
Object
    Collection
        SequencableCollection
            ArrayedCollection
                RunArray
                    WeightedCollection
```

WeightedCollection methods for: growing

The **grow:** message allows the size of a WeightedCollection to be changed, while still retaining the underlying collection and relative probabilities. The protocol inherited from RunArray allows the underlying values collection to be altered, should the user see fit.

Note that the weight and run sets should be changed only at the user's peril, since these are not mutually constrained in this implementation of WeightedCollection.

WeightedCollection methods for: growing

```
grow: size
    "Transform ourself into a new weighted collection..."

    | new |
    new ← WeightedCollection
            on: self values
            withSize: size
```

```
                andWeights: self weights.
    ^self become: new
```

WeightedCollection methods for: private

Access to the weight collection is provided for the purpose of internal abstraction only. Users should not attempt to alter the weight set of a Weighted collection once it is created.

WeightedCollection methods for: private

```
weights
    "Return the weights collection..."

    ^weights

weights: aCollection
    "Save the weights collection..."

    ^weights ← aCollection
```

WeightedCollection class

class	WeightedCollection class
superclass	RunArray class

WeightedCollection class methods for: instance creation

The **on:withSize:andWeights:** method is the only means by which a WeightedCollection should be created. (The **new** method should probably be overridden to enforce this restriction.) It works as follows: First, a collection consisting of the desired collection size scaled by each weight is constructed. (Note that the values are rounded to the nearest Integer.) Next, these values are summed. The *error* variable is then assigned the difference between this total and the desired collection size.

If the sum of the counts produced by weighting the size comes up short, then *error* will be positive. If this value is too large, *error* will be negative. The test on error is used to set *adjust* to a positive count of the number of values that need to be changed to balance the counts, and *increment* to plus or minus one. The error is then distributed among the first *adjust* elements of the *counts* collection. This collection then appeals to RunArray class to construct our underlying RunArray. The original weight set is also stored for use by **grow:**.

WeightedCollection class methods for: instance creation

```
on: aCollection withSize: size andWeights: weights
    "Create a new weighted collection..."

    | counts total error adjust increment new |
    counts ← weights collect: [:weight | (weight * size) rounded asInteger].
    total ← counts inject: 0 into: [:sum :count | sum + count].
    error ← size - total.
    error > 0
        ifTrue: [adjust ← error. increment ← 1]
        ifFalse: [adjust ← error negated. increment ← -1].
    (1 to: adjust)
        do: [:i | counts at: i put: (counts at: i) + 1].
    new ← super runs: counts values: aCollection.
    new weights: weights.
    ^new
```

WeightedCollection class methods for: examples

This example show how one might construct a ten element weighted collection over the Array shown. The result is a collection with the following structure: #(dog dog dog dog cat cat cat mouse mouse mouse).

WeightedCollection class methods for: examples

```
example
    "WeightedCollection example inspect"

    ^WeightedCollection
        on: #(dog cat mouse )
        withSize: 10
        andWeights: #(0.3 0.3 0.3 )
```

SequenceGenerator

Class SequenceGenerator is used to construct a stimulus sequence given a fixed base collection and a given size and set of relative probabilities. The present implementation of SequenceGenerator uses a WeightedCollection to do most of its work. WeightedCollection is, in fact, a generalization of an earlier implementation of SequenceGenerator. SequenceGenerator still adds a constraint check that ensures that the given probability set adds up to one. This check could (should) probably be moved to WeightedCollection as well.

class name SequenceGenerator
superclass Object
instance variable names
length
probabilities
stimuli
class variable names
pool dictionaries
category Sequence-Support
hierarchy
Object
SequenceGenerator

SequenceGenerator methods for: sequence construction

The **generateSequenceOn:withSize:andProbabilities:** method constructs a stimulus sequence with the desired base set, size and probabilities and returns it to its caller. We first store the parameters given us, then do a constraint test on the probability set, and then construct and return the stimulus sequence.

SequenceGenerator methods for: sequence construction

```
generateSequenceOn: stims withSize: size andProbabilities: probs
    "Save our parameters, constrain the probability set, and build a
    sequence..."

    length ← size.
    probabilities ← probs.
    stimuli ← stims.
    self constrainProbabilities.
    ^self constructSequence
```

SequenceGenerator methods for: private

The **constrainProbabilities** method makes sure that the probability set given us sums to one. It will adjust the final value if it finds a minor discrepancy. (Note that for this reason, we should probably have done a copy on the set before storing it in the instance variable above.) If a major problem is found, an error is declared.

The **constructSequence** method merely passes the buck to WeightedCollection to get its work done.

```
SequenceGenerator methods for:  private

    constrainProbabilities
        "Make sure the probabilities add up to 1.0..."

        | sum last |
        sum ← (1 to: probabilities size - 1)
                    inject: 0 into: [:sum :i | sum ← sum + (probabilities at: i)].
        (sum < 0 or: [sum > 1])
            ifTrue: [self error: 'Bad probability sum...'].
        probabilities at: probabilities size put: (1.0-sum)

    constructSequence
        "Build a weighted collection given our current size and probabilities... "

        ^WeightedCollection on: stimuli withSize: length andWeights: probabilities

    "Notes 6/23/86:  Hope they allow zero length runs..."
    "Notes 7/24/86:  They do...."
    "Notes 8/25/86:  Changed over to WeightedCollections..."
```

SequenceGenerator class

class	SequenceGenerator class
superclass	Object class

SequenceGenerator class methods for: examples

The **example1** method shows a typical application of a SequenceGenerator.

SequenceGenerator class methods for: examples

example1

```
"SequenceGenerator example1 inspect"

| s stims probs |
s ← SequenceGenerator new.
stims ← #(catA catB ).
probs ← Array with: 0.5 with: 0.5.
^s
    generateSequenceOn: stims
    withSize: 100
    andProbabilities: probs
```

Response-Support

The Response-Support system category represents an effort to model the hardware and software components of the subject response reporting system as concrete objects. The rationale for this is similar to that for making the Stimulus-Generators concrete objects. That is to say, it was hoped that embodying the response collection strategies we use in separate, discrete components, we might be able to modify these strategies from application to application merely by dropping in a new component. The classes in this category are:

ButtonBox	Simulates a subject response box
ButtonBoxResponse	Describes a subject response instance

ButtonBox

The ButtonBox class models a two button subject response box. Such a response box typically will report button presses using a pair of digital input bits. Hence, ButtonBoxes contain references to a pair of simulated input bits.

A clock is also associated with each ButtonBox for response time reporting. Since InputBit events are simulated using the Smalltalk mouse buttons, it is possible to demonstrate actual ButtonBox events when a simulated BatteryItem is run.

class name ButtonBox
superclass Object
instance variable names
  bitA
  bitB
  clock
class variable names
pool dictionaries
category Response-Support
hierarchy
  Object
    ButtonBox
    ButtonBoxResponse

ButtonBox methods for: accessing

The *bitA* and *bitB* instance variables contain references to the InputBits that we use to report responses. The *clock* instance variable is used for response time reporting.

The **enable** method arms the ButtonBox by passing an **enableUsingClock:** message to both of the ButtonBox's InputBits.

The **response** query is sent by the user of a ButtonBox to test for whether any response activity has taken place. A ButtonBoxResponse object will always be the result of such a query. If no button press has occurred since the last query, the ButtonBoxResponse object will contain only default values.

Responses are detected by testing the flags of each InputBit object. If one of these flags is set, a #buttonA or #buttonB response event is reported, together with the time (relative to the clock that was originally passed) at which the event occurred. Note that if both button A and button B were pressed, the button A event is given precedence. More complex button box designs might adopt some different strategy.

ButtonBox methods for: accessing

```
bitA
    ^bitA

bitA: argument
    bitA ← argument.
    ^argument

bitB
    ^bitB

bitB: argument
    bitB ← argument.
    ^argument

clock
    ^clock

clock: argument
    clock ← argument.
    ^argument

enable
    "Enable both our input bits using our clock..."

    self bitA enableUsingClock: self clock.
    self bitB enableUsingClock: self clock

response
    "Return a response object with the bit for which we recorded
    a response (if any) and the response time for it..."

    | resp |
    resp ← ButtonBoxResponse new.
    (self bitA flag) ifTrue:
        [ resp responseButton: #buttonA. resp responseTime: self bitA time. ^resp].
    (self bitB flag) ifTrue:
        [ resp responseButton: #buttonB. resp responseTime: self bitB time. ^resp].
    ^resp
```

ButtonBox class

class ButtonBox class
superclass Object class

ButtonBox class methods for: instance creation

The **bitA:bitB:usingClock:** method creates a ButtonBox and assigns it the indicated realtime resources. As with several other instance creation methods seen herein, we might want to consider enforcing the convention that this method be used to create ButtonBoxes by overriding **new**.

ButtonBox class methods for: instance creation

```
bitA: a bitB: b usingClock: aClock
    "Create a new button box..."

    | box |
    box ← super new.
    box bitA: (InputBit onBit: a).
    box bitB: (InputBit onBit: b).
    box clock: aClock.
    ^box
```

ButtonBoxResponse

ButtonBoxResponse objects are returned by ButtonBoxes in response to response queries. They convey a response button code and a response time. The possible response button codes are determined by the ButtonBox object.

class name ButtonBoxResponse
superclass Object
instance variable names
responseButton
responseTime
class variable names
pool dictionaries
category ResponseSupport
hierarchy
Object
ButtonBox
ButtonBoxResponse

ButtonBoxResponse methods for: accessing

ButtonBoxResponse methods for: accessing

responseButton
^responseButton

responseButton: argument
responseButton ← argument.
^argument

responseTime
^responseTime

responseTime: argument
responseTime ← argument.
^argument

Data-Management

The Data-Management system category contains the definitions for a set of record-like objects that help to implement the battery data management scheme. These classes are:

BatteryBlock	A per-block data record
BatteryTrial	A per-trial data record
DataDictionary	Repository for archived battery data
DataDictionaryEntry	An entry in a DataDictionary

BatteryBlock

BatteryBlock object tie together all the information that a battery item produces for each block it runs. A BatteryBlock stores the block average collection, a copy of the block parameters, the single trial collection for the block, and the block tally set.

class name BatteryBlock
superclass Object
instance variable names
parameters
singleTrials
tallies
averages
class variable names
pool dictionaries
category Data-Management
hierarchy
Object
BatteryBlock

BatteryBlock methods for: accessing

BatteryBlock methods for: accessing

```
averages
    ^averages

averages: argument
    averages ← argument.
    ^argument

parameters
    ^parameters

parameters: argument
    parameters ← argument.
    ^argument

singleTrials
    ^singleTrials

singleTrials: argument
    singleTrials ← argument.
    ^argument

tallies
    ^tallies
```

tallies: argument

```
tallies ← argument.
^argument
```

BatteryTrial

BatteryTrial objects tie together the data produced for each trial. A block's single trial collection is a collection of these objects, one for each trial. The following data are kept for each trial: A copy of the digitized data, the subject's response time (if any), a subject response type, the stimulus descriptor for the trial, and the trial number.

```
class name      BatteryTrial
superclass      Object
instance variable names
                trialNumber
                stimulus
                responseType
                responseTime
                data
class variable names
pool dictionaries
category        Data-Management
hierarchy
                Object
                    BatteryTrial
```

BatteryTrial methods for: accessing

BatteryTrial methods for: accessing

```
data
    ^data

data: argument
    data ← argument.
    ^argument

responseTime
    ^responseTime

responseTime: argument
    responseTime ← argument.
    ^argument

responseType
    ^responseType

responseType: argument
    responseType ← argument.
    ^argument
```

stimulus
^stimulus

stimulus: argument

```
stimulus ← argument.
^argument
```

trialNumber

```
^trialNumber
```

trialNumber: argument

```
trialNumber ← argument.
^argument
```

DataDictionary

DataDictionary objects are subclasses of AccessableDictionaries. They add no additional protocol. A single instance of DataDictionary is kept in a BatteryItem class variable as a global repository for battery data.

class name DataDictionary
superclass AccessableDictionary
instance variable names
tally
class variable names
pool dictionaries
category Data-Management
hierarchy

```
Object
    Collection
        Set
            Dictionary
                AccessableDictionary
                    DataDictionary
```

DataDictionaryEntry

DataDictionaryEntry objects play the role of file system directory entries in the battery data management scheme. They record the following: some data (which will always be a BatteryBlock given the current battery simulation), the date and time of the entries creation, a label string of some sort, an entry type field, and an entry name.

class name DataDictionaryEntry
superclass Object
instance variable names
name
type
label
date
time
data
class variable names
pool dictionaries
category Data-Management
hierarchy
Object
DataDictionaryEntry

DataDictionaryEntry methods for: accessing

DataDictionaryEntry methods for: accessing

data
^data

data: argument
data ← argument.
^argument

date
^date

date: argument
date ← argument.
^argument

label
^label

label: argument
label ← argument.
^argument

name
^name

name: argument
name ← argument.
^argument

time
^time

time: argument
time ← argument.
^argument

type
^type

type: argument
type ← argument.
^argument

DataDictionaryEntry class

class	DataDictionaryEntry class
superclass	Object class

DataDictionaryEntry class methods for: instance creation

The method given below provides a convenient shorthand for creating DataDictionaryEntry objects. Note in particular that the date and time time stamp is generated automatically by this method.

DataDictionaryEntry class methods for: instance creation

```
entryNamed: aString withLabel: aLabel andData: anObject
    "Create and entry and fill it in as indicated, together with the current
    date and time..."

    | entry date |
    entry ← super new.
    entry name: aString.
    entry label: aLabel.
    entry data: anObject.
    date ← Date dateAndTimeNow.
    entry date: (date at: 1).
    entry time: (date at: 2).
```

^entry

Interface-Battery

The classes defined in the Interface-Battery category implement the user interface for the Smalltalk battery simulation. The classes in this category are:

ListHolder	A simple generic model for ListViews
ItemListController	Controller for a battery item list
ParameterListController	Controller for a parameter list
BatteryCodeController	Controller for the expression window
WaveformController	Controller for a waveform view
BatteryBrowser	Model for a set of battery items
ItemListView	View of a list of battery items
ParameterListView	View of the parameter list
BatteryView	TopView for a battery browser
BatteryCodeView	View for expression evaluation
WaveformView	A graphical view of a waveform

ListHolder

ListHolder objects provide a service similar to that provided by StringHolders. That is, they serve as a generic model for simple list objects.

This function has been largely subsumed by Version II SelectionInListView objects. The Apple Smalltalk image used for this simulation was based on a Xerox Smalltalk-80 Version I image that did not have pluggable views.

class name ListHolder
superclass AccessableObject
instance variable names
items

list
listIndex
class variable names
pool dictionaries
category Interface-Battery
hierarchy

```
Object
    AccessableObject
        ListHolder
```

ListHolder methods for: view access

These methods implement the default interface between a list model and its ListView.

ListHolder methods for: view access

```
list
    "Return our list..."

    ^list

list: argument
    "Set the new list and notify our dependents..."

    list ← argument.
    self changed: #list.
    ^argument

listIndex
    "Return our copy of the list index..."
```

^listIndex

listIndex: argument

"Reset the list index, and notify our dependents..."

```
listIndex ← argument.
self changed: #listIndex.
^argument
```

ListHolder methods for: controller access

These methods implement the default interface between a list model and its ListController.

ListHolder methods for: controller access

toggleListIndex: index

"Just keep our copy of the list index up to date..."

```
index = self listIndex
    ifTrue: [self listIndex: 0]
    ifFalse: [self listIndex: index]
```

ListHolder class

class	ListHolder class
superclass	AccessableObject class

ListHolder class methods for: instance creation

ListHolder class methods for: instance creation

on: list

"Create a new ListHolder for the designated list..."

```
| aListHolder |
aListHolder ← self new.
aListHolder list: list.
^aListHolder
```

ItemListController

ItemListControllers operate the battery browser's battery item list.

class name ItemListController
superclass ListController
instance variable names
model
view
sensor

redButtonMenu
redButtonMessages
yellowButtonMenu
yellowButtonMessages
blueButtonMenu
blueButtonMessages

scrollBar
marker
savedArea
class variable names
ItemListYellowButtonMessages
ItemListYellowButtonMenu
category Interface-Battery
hierarchy

```
Object
    Controller
        MouseMenuController
            ScrollController
                ListController
                    ItemListController
```

ItemListController methods for: instance initialization

Make sure new ItemListControllers have their yellowButtonMenus set up correctly. The menu/message sets are kept in a pair of class variables for this purpose.

ItemListController methods for: instance initialization

```
initialize
    "Set up the yellow button menu, among other things..."

    super initialize.
    self initializeYellowButtonMenu

initializeYellowButtonMenu
```

```
self yellowButtonMenu: ItemListYellowButtonMenu
    yellowButtonMessages: ItemListYellowButtonMessages
```

ItemListController methods for: menu messages

The **doNothing** method illustrates where additional menu messages would fit were one to want to add them.

The **changeModelSelection:** method inherited from ListController is overridden here to invoke **toggleItemListIndex** instead of **toggleListIndex**. When complex models like the BatteryBrowser with multiple cooperating views are used, it is necessary to differentiate at the model level among list update requests. In conventional ListViews, this is usually done by overriding **changeModelSelection** in the manner just described.

Version II SelectionInListViews allow the selectors for these sorts of functions to be built into "adaptor" variables stored in the Views. The views themselves must also distinguish multiple update requests when a complex model is used. This is done using view defined **update:** parameter protocols. Traditional ListViews must override **update:** to add this behavior. SelectionInListViews once again make this part of the adaptor.

ItemListController methods for: menu messages

```
doNothing
    "Do as it says, for testing...."
```

ItemListController methods for: private

ItemListController methods for: private

```
changeModelSelection: anInteger

    model toggleItemListIndex: anInteger
```

ItemListController class

class ItemListController class
subclass ListController class

ItemListController class methods for: class initialization

The default menu and message lists are stored in class variables so that they need not be copied of every time one of these controllers is created.

ItemListController class methods for: class initialization

```
initialize
    "ItemListController initialize"

    ItemListYellowButtonMenu ← PopUpMenu labels: 'do nothing
still do nothing' lines: #(1 ).
    ItemListYellowButtonMessages ← #(doNothing doNothing )

ItemListController initialize
```

ParameterListController

ParameterListControllers implement the controller end of the MVC triad for the battery parameter list view.

class name ParameterListController
superclass ListController
instance variable names
model
view
sensor

redButtonMenu
redButtonMessages
yellowButtonMenu
yellowButtonMessages
blueButtonMenu
blueButtonMessages

scrollBar
marker
savedArea
class variable names
ParameterListYellowButtonMessages
ParameterListYellowButtonMenu
pool dictionaries
category Interface-Battery
hierarchy
Object
    Controller
        MouseMenuController
            ScrollController
                ListController
                    ParameterListController

ParameterListController methods for: instance initialization

The default menu and message lists are copied from two of the ParameterListController's class variables.

ParameterListController methods for: instance initialization

initialize

```
"Set up the yellow button menu, among other things..."

super initialize.
self initializeYellowButtonMenu
```

initializeYellowButtonMenu

```
self yellowButtonMenu: ParameterListYellowButtonMenu
    yellowButtonMessages: ParameterListYellowButtonMessages
```

ParameterListController methods for: menu messages

As with ItemListController, **changeModelSelection** is overridden so that it passes toggle parameter list requests to this object's model in such a way so that the model can learn their origin.

ParameterListController methods for: menu messages

doNothing
"Do as it says, for testing...."

ParameterListController methods for: private

ParameterListController methods for: private

changeModelSelection: anInteger

```
model toggleParameterListIndex: anInteger
```

ParameterListController class

class ParameterListController class
superclass ListController class

ParameterListController class methods for: class initialization

Two menu entries are defined (as examples) here, both of which are linked to the **doNothing** method.

ParameterListController class methods for: class initialization

initialize
"ParameterListController initialize"

```
    ParameterListYellowButtonMenu ← PopUpMenu labels: 'do nothing
still do nothing' lines: #(1 ).
```

```
	ParameterListYellowButtonMessages ← #(doNothing doNothing )

ParameterListController initialize
```

BatteryCodeController

The BatteryCodeController class adds an accept message that forwards the results of the accept back to the model using **replaceParameterSelectionValue:**.

class name BatteryCodeController
superclass StringHolderController
instance variable names

model
view
sensor

redButtonMenu
redButtonMessages
yellowButtonMenu
yellowButtonMessages
blueButtonMenu
blueButtonMessages

scrollBar
marker
savedArea

paragraph
startBlock
stopBlock
beginTypeInBlock
emphasisHere
initialText
selectionShowing

isLockingOn

class variable names
pool dictionaries
category Interface-Battery
hierarchy

```
Object
    Controller
        MouseMenuController
            ScrollController
                ParagraphEditor
                    StringHolderController
                        BatteryCodeController
```

BatteryCodeController methods for: menu messages

The **accept** method evaluates the BatteryCodeController's paragraph's string and makes the result the new parameter selection value of its model.

BatteryCodeController methods for: menu messages

accept

```
| result |
(model isUnlocked or: [model selectionUnmodifiable])
    ifTrue: [^view flash].
self controlTerminate.
result ← model doItReceiver class evaluatorClass new
            evaluate: (ReadStream on: paragraph string)
            in: model doItContext
            to: model doItReceiver
            notifying: self
            ifFail:  [self controlInitialize. ^nil].
result == #failedDoit
    ifFalse:
        [model replaceParameterSelectionValue: result.
        self selectFrom: 1 to: paragraph text size.
        self deselect.
        self replaceSelectionWith: result printString asText.
        self selectAt: 1.
        super accept].
self controlInitialize
```

WaveformController

WaveformController objects provide cursor control for WaveformView objects. When the mouse is within the graphBox of a WaveformView, the cursor is turned into a cross hair cursor, and the WaveformView's coordinate display is updated.

class name WaveformController
superclass Controller
instance variable names
model
view
sensor

cursorPosition
class variable names
pool dictionaries
category Interface-Battery
hierarchy
Object
Controller
WaveformController

WaveformController methods for: controlling

To control activity, the WaveformController must see if the view's graphBox contains the cursorPoint. If so, the cursor is changed to a cross hair cursor and the waveform coordinate display is updated. If the cursor has left the graphBox, the normal cursor is restored.

WaveformController methods for: controlling

```
controlActivity
    (view graphBox containsPoint: sensor cursorPoint )
        ifTrue: [self controlCursor]
        ifFalse: [Cursor normal show]

controlCursor
    | point |
    Cursor crossHair show.
    self view updateText.
    point ← sensor cursorPoint.
    self cursorPosition: point
```

"Brian Foote 28 July 1987 Added code to copy point to cursorPosition..."

controlInitialize

"Say there is no cursor position for now..."

```
self cursorPosition: nil
```

controlTerminate

"Get rid of the cross hair cursor..."

```
Cursor normal show
```

WaveformController methods for: accessing

WaveformController methods for: accessing

cursorPosition

```
^cursorPosition
```

cursorPosition: argument

```
cursorPosition ← argument.
^argument
```

BatteryBrowser

BatteryBrowser objects are the common models that tie together the components of a BatteryView.

class name BatteryBrowser
superclass StringHolder
instance variable names
contents
isLocked

itemList
itemListIndex
parameterList
parameterListIndex
runSwitch
parameterKeyList
waveformView
class variable names
pool dictionaries
category Interface-Battery
hierarchy

```
Object
    StringHolder
        BatteryBrowser
```

BatteryBrowser methods for: instance initialization

The **initialize** method starts us out with no item or parameter selections.

BatteryBrowser methods for: instance initialization

```
initialize
    "Set a few defaults..."

    self itemListIndex: 0.
    self parameterListIndex: 0
```

BatteryBrowser methods for: accessing

BatteryBrowser methods for: accessing

```
itemList
    ^itemList

itemList: argument
itemList ← argument.
```

^argument

itemListIndex
  ^itemListIndex

itemListIndex: argument
  itemListIndex ← argument.
  ^argument

parameterKeyList
  ^parameterKeyList

parameterKeyList: argument
  parameterKeyList ← argument.
  ^argument

parameterList
  ^parameterList

parameterList: argument
  parameterList ← argument.
  ^argument

parameterListIndex
  ^parameterListIndex

parameterListIndex: argument
  parameterListIndex ← argument.
  ^argument

runSwitch
  ^runSwitch

runSwitch: argument
  runSwitch ← argument.
  ^argument

waveformView
  ^waveformView

waveformView: argument
  waveformView ← argument.
  ^argument

BatteryBrowser methods for: list access

The **replaceParameterSelectionValue** method is invoked by BatteryCodeControllers when an accept is performed. It replaces the value of the currently selected parameter with a new value.

The **toggleItemListIndex:** method is invoked when a new item list selection is made. First, the wait cursor is turned on. Then our item list selection is updated. Next, since either the current battery item was deselected or a new battery item in the item list was selected, the parameter list must be reset. We update the parameter list and the parameter key list, set its selection to zero, and clear the contents of our code view. We then declare that the item list has changed. (ParameterListViews respond to both item list and parameter list changes. BrowserCodeControllers respond to any update whatsoever.)

The **toggleParameterListIndex:** method saves the new list index given it, and stores the store string for the current parameter value as our contents. A parameter list update is then declared, which will update the ParameterListView and the BatteryCodeView.

BatteryBrowser methods for: list access

```
replaceParameterSelectionValue: value
    "Set the current parameter to the indicated value..."

    | key parms |
    key ← self parameterKeyList at: self parameterListIndex.
    parms ← self currentItem parameters.
    parms at: key put: value.
    self parameterList: self currentParameterList

toggleItemListIndex: anInteger
    "Toggle our item list index..."

    Cursor wait
        showWhile:
            [self itemListIndex: (self itemListIndex = anInteger
                        ifTrue: [0]
                        ifFalse: [anInteger]).
            self parameterList: self currentParameterList.
            self parameterKeyList: self currentParameterKeyList.
            self parameterListIndex: 0.
            self contents: ''.
            self changed: #itemList]

toggleParameterListIndex: anInteger
    "Toggle our parameter list index..."

    self parameterListIndex: (self parameterListIndex = anInteger
            ifTrue: [0]
            ifFalse: [anInteger]).
    self contents: self currentParameterValue storeString.
    self changed: #parameterListIndex
```

BatteryBrowser methods for: switch access

The run switch object sends us this message when the run switch is pressed. If a battery item is selected, it is started up by sending it a **doBlock** message. When the block finally finishes, the run switch is turned off.

```
BatteryBrowser methods for:  switch access

        run
            "Off we go..."

            self  currentItem ~= nil ifTrue: [self  currentItem doBlock].
            self runSwitch turnOff
```

BatteryBrowser methods for: selecting

The **selectionUnmodifiable** query is sent by **accept** method in BatteryCodeController to its models. Hence, BatteryBrowser must implement this method. It returns true when there is no current selection. (Note that the **accept** method used in BatteryCodeController was borrowed from the one found in the Version I InspectCodeController.)

```
BatteryBrowser methods for:  selecting

        selectionUnmodifiable
            "If the selection is zero, forget it..."

            ^parameterListIndex=0
```

BatteryBrowser methods for: private

The methods in this protocol category encapsulate the BatteryBrowser's internal schemes for generating various structures that are used by other parts of the BatteryBrowser.

The **currentItem** method uses the itemListIndex to return either nil or the current battery item itself.   The **currentItemList** is constructed by asking each battery item in our item list to ask its parameter object for an item name. These names are collected and returned as the current item list.

The **currentParameterKeyList** method returns an OrderedCollection containing the keys for the current list of parameters. It derives this by calling the **currentParameterList** method. This method returns an empty AccessableDictionary if no item is currently selected. If an item is selected, we fetch its parameter object (which is an AccessableObject) and turn this into an AccessableDictionary, which we then return.)

The **currentParameterValue** method uses the key list to index the parameter list which in turn allows us access to a parameter value.

BatteryBrowser methods for: private

```
currentItem
    "Return the current battery item..."

    self itemListIndex = 0
        ifTrue: [^nil]
        ifFalse: [^self itemList at: itemListIndex]

currentItemList
    "Get the current item list..."

    ^self itemList collect: [:anItem | anItem parameters name].

currentParameterKeyList
    "Get the current parameter key list..."

    ^self currentParameterList keys asOrderedCollection

currentParameterList
    "Get the current parameter list..."

    | item parms |
    self currentItem = nil ifTrue: [^AccessableDictionary new].
    item ← self currentItem.
    parms ← item parameters asAccessableDictionary.
    ^parms

currentParameterValue
    "Return the current parameter value..."

    self parameterListIndex = 0
        ifTrue: [^nil]
```

ifFalse: [^self parameterList at: (self parameterKeyList at: self parameterListIndex)]

BatteryBrowser class

class BatteryBrowser class
superclass StringHolder class

BatteryBrowser class methods for: instance creation

The **openOn:** message creates a BatteryBrowser given a list of BatteryItem objects. It initializes the browser, sets it list as indicated, and then informs every item in the list that they belong to this browser.

BatteryBrowser class methods for: instance creation

```
openOn: aList
    "BatteryBrowser openOn: BatteryItem defaultItemList"

    | browser |
    browser ← self new.
    browser initialize.
    browser itemList: aList.
    aList do: [:item | item parameters browser: browser].
    ^browser
```

ItemListView

ItemListView objects merely specialize ListView objects by designating that the default controller class is ItemListController rather than ListController. (Note that Version II SelectionInListViews use an "adaptor" that makes this specialization unnecessary.)

class name ItemListView
superclass ListView
instance variable names
model
view
superView
subViews
transformation
viewport
window
displayTransformation
insetDisplayBox
borderWidth
borderColor
insideColor
boundingBox

list
selection
topDelimiter
bottomDelimiter
lineSpacing
isEmpty
class variable names
pool dictionaries
category Interface-Battery
hierarchy
Object
    View
        ListView
            ItemListView

ItemListView methods for: controller access

ItemListView methods for: controller access

defaultControllerClass
"Give a default controller for us..."

^ItemListController

ParameterListView

ParameterListView objects specialize ListView objects by designating that the default controller class is ItemListController rather than ListController. They also provide **update:** protocol for updating the parameter list in response to both item list and parameter list change requests.

class name ParameterListView
superclass ListView
instance variable names
model
view
superView
subViews
transformation
viewport
window
displayTransformation
insetDisplayBox
borderWidth
borderColor
insideColor
boundingBox

list
selection
topDelimiter
bottomDelimiter
lineSpacing
isEmpty
class variable names
pool dictionaries
category Interface-Battery
hierarchy
Object
    View
        ListView
            ParameterListView

ParameterListView methods for: controller access

We have our own controller (which does nothing useful) so we designate it as our default.

ParameterListView methods for: controller access

defaultControllerClass

```
"Give a default controller for us..."

^ParameterListController
```

ParameterListView methods for: updating

When a ParameterListView receives an *#itemList* update request, it resets its parameter list and redisplay itself. When its gets a *#parameterListIndex* update request, it asks itself to move the selection box.

ParameterListView methods for: updating

```
update: anObject
    "Something changed..."

    anObject == #itemList
        ifTrue:
            [self list: model parameterKeyList.
            selection ← model parameterListIndex.
            self displayView].
    anObject == #parameterListIndex
        ifTrue:
            [self moveSelectionBox: model parameterListIndex]
```

BatteryView

BatteryView objects provide the StandardSystemView for BatteryBrowsers. It is the BatteryView that creates and composes the subViews that comprises the BatteryBrowser.

class name BatteryView
superclass StandardSystemView
instance variable names
model
view
superView
subViews
transformation
viewport
window
displayTransformation
insetDisplayBox
borderWidth
borderColor
insideColor
boundingBox

labelFrame
labelText
isLabelComplemented
savedSubViews
minimumSize
maximumSize
collapsedViewport
expandedViewport
labelBits
windowBits
class variable names
pool dictionaries
category Interface-Battery
hierarchy
Object
View
StandardSystemView
BatteryView

BatteryView class

class BatteryView class
superclass StandardSystemView class

BatteryView class methods for: instance creation

To start up a BatteryBrowser, one should first create a BatteryBrowser on a list of battery items, then pass this browser as the argument to an **openOn:** message to BatteryView. The comment below gives an example of this.

BatteryView class methods for: instance creation

```
openOn: aBrowser
    "BatteryView openOn: (BatteryBrowser openOn: BatteryItem defaultItemList)"

    | topView itemList itemView parmList parmView stringView holder
        switch wave switchView waveView |
    topView ← self new.
    topView model: nil; label: 'Battery Item Browser'; minimumSize: 100 @ 100.
    topView borderColor: Form lightGray; insideColor: Form gray.
    topView borderWidth: 0.
    itemView ← ItemListView new.
    itemList ← aBrowser itemList collect: [:anItem | anItem parameters name].
    itemView model: aBrowser.
    itemView list: itemList.
    itemView
        borderWidthLeft: 2
        right: 1
        top: 2
        bottom: 1.
    itemView window: (0 @ 0 extent: 200 @ 100).
    topView addSubView: itemView.
    parmView ← ParameterListView new.
    parmView model: aBrowser.
    parmView
        borderWidthLeft: 1
        right: 1
        top: 2
        bottom: 1.
    parmView window: (0 @ 0 extent: 200 @ 100).
    topView
        addSubView: parmView
        align: parmView viewport topLeft
        with: itemView viewport topRight.
    aBrowser contents: ''.
```

```
stringView ← BatteryCodeView container: aBrowser.
stringView window: (0 @ 0 extent: 200 @ 70).
```

```
    stringView
        borderWidthLeft: 1
        right: 2
        top: 2
        bottom: 1.
    topView
        addSubView: stringView
        align: stringView viewport topLeft
        with: parmView viewport topRight.
    switch ← Switch newOff.
    switch onAction: [aBrowser run].
    aBrowser runSwitch: switch.
    switchView ← SwitchView new.
    switchView model: switch; label: 'Run' asParagraph.
    switchView
        borderWidthLeft: 1
        right: 2
        top: 1
        bottom: 1.
    switchView window: (0 @ 0 extent: 200 @ 30).
    topView
        addSubView: switchView
        align: switchView viewport bottomLeft
        with: parmView viewport bottomRight.
    waveView ← WaveformView new.
    aBrowser waveformView: waveView.
    wave ← Waveform points: 10.
    wave zero.
    waveView model: wave; window: (0 @ 0 extent: 600 @ 250); insideColor: Form
white.
    waveView
        borderWidthLeft: 2
        right: 2
        top: 1
        bottom: 2.
    waveView dataWindow: (1 @ 2500 corner: wave size @ -2500).
    topView
        addSubView: waveView
        align: waveView viewport topLeft
        with: itemView viewport bottomLeft.
    topView controller open
```

BatteryCodeView

BatteryCodeView objects provide views of the values of currently selected parameters.

class name BatteryCodeView
superclass StringHolderView
instance variable names
model
view
superView
subViews
transformation
viewport
window
displayTransformation
insetDisplayBox
borderWidth
borderColor
insideColor
boundingBox

displayContents
class variable names
pool dictionaries
category Interface-Battery
hierarchy

```
Object
    View
        StringHolderView
            BatteryCodeView
```

BatteryCodeView methods for: controller access

We are a pre-pluggable view code view that exists only to designate a default controller.

BatteryCodeView methods for: controller access

defaultControllerClass

```
^BatteryCodeController
```

WaveformView

WaveformView objects generate graphical waveform displays. They also allow a cursor to be driven across these waveforms, and display information about its position.

class name WaveformView
superclass View
instance variable names

model
view
superView
subViews
transformation
viewport
window
displayTransformation
insetDisplayBox
borderWidth
borderColor
insideColor
boundingBox

dataWindow
dataBox
dataTransformation
dataInsets
graphBox
length
interval
thickness
topMargin
bottomMargin
leftMargin
rightMargin
quill
title
titleEmphasis
lastPosition

class variable names
pool dictionaries
category Interface-Battery
hierarchy

Object
    View
        WaveformView

WaveformView methods for: instance initialization

WaveformView methods for: instance initialization

initialize

```
"Initialize this instance..."

self thickness: 5 @ 5.
self length: 5 @ 5.
self interval: 25 @ 1000.
self dataInsets: 0.0 @ 0.1.
self topMargin: 30.
self bottomMargin: 50.
self leftMargin: 50.
self rightMargin: 10.
self quill: Pen new.
self title: 'a Waveform View'.
self titleEmphasis: 5.
self lastPosition: 0.99 @ 0.99.
super initialize
```

WaveformView methods for: accessing

WaveformView methods for: accessing

bottomMargin

```
^bottomMargin
```

bottomMargin: argument

```
bottomMargin ← argument.
^argument
```

dataBox

```
^dataBox
```

dataBox: argument

```
dataBox ← argument.
^argument
```

dataInsets

```
^dataInsets
```

dataInsets: argument

```
dataInsets ← argument.
^argument
```

dataTransformation

```
^dataTransformation
```

dataTransformation: argument
dataTransformation ← argument.
^argument

dataWindow
^dataWindow

dataWindow: argument
dataWindow ← argument.
^argument

graphBox
^graphBox

graphBox: argument
graphBox ← argument.
^argument

interval
^interval

interval: argument
interval ← argument.
^argument

lastPosition
^lastPosition

lastPosition: argument
lastPosition ← argument.
^argument

leftMargin
^leftMargin

leftMargin: argument
leftMargin ← argument.
^argument

length
^length

length: argument
length ← argument.
^argument

quill
^quill

quill: argument
quill ← argument.
^argument

rightMargin
```
^rightMargin
```

rightMargin: argument
```
rightMargin ← argument.
^argument
```

thickness
^thickness

thickness: argument
thickness ← argument.
^argument

title
^title

title: argument
title ← argument.
^argument

titleEmphasis
^titleEmphasis

titleEmphasis: argument
titleEmphasis ← argument.
^argument

topMargin
^topMargin

topMargin: argument
topMargin ← argument.
^argument

WaveformView methods for: controller access

WaveformView methods for: controller access

defaultControllerClass
"Use a WaveformController to do our dirty work..."

^WaveformController

WaveformView methods for: displaying

The messages in this protocol category perform the tasks associated with displaying waveforms on the screen.

The **clear** method fills our inset display box with our insideColor. The **displayAbscissa** method draws an X axis.

The **displayData** method plots our model (a Waveform).

The **displayOrdinate** method draws our Y axis.

The **displayTitle** method draws our title.

The **displayView** method is the standard method for drawing a view. It resets our variables, clears us, and draws our components.

The **displayXTick:** and **displayYTick:** methods are used for tick mark generation.

The **resetVariables** method resets the reference boxes that we use to draw a waveform graph.

WaveformView methods for: displaying

clear

```
"Zap the display box..."

Display fill: self insetDisplayBox mask: self insideColor
```

displayAbscissa

```
"Draw the X axis..."

| box x start stop increment y |
x ← self graphBox left - self thickness y.
y ← self graphBox bottom + self thickness x.
box ← x @ self graphBox bottom corner: self graphBox right @ y.
Display
    fill: box
    rule: Form over
    mask: Form black.
increment ← interval x.
start ← self dataWindow left.
start ← start roundUpTo: increment.
stop ← self dataWindow right.
stop ← stop roundDownTo: increment.
start ~= self dataWindow left ifTrue: [self displayXTick: self dataWindow left].
(start to: stop by: increment)
    do: [:value | self displayXTick: value]
```

displayData

```
"Draw the waveform..."

| pen point |
Display fill: self graphBox mask: Form white.
pen ← Pen new.
pen frame: self graphBox.
point ← Point x: 1 y: (model at: 1).
pen place: (self dataTransformation applyTo: point).
(2 to: model size)
    do:
        [:i |
        point ← Point x: i y: (model at: i).
        point ← self dataTransformation applyTo: point.
        pen goto: point]
```

displayOrdinate

```
"Draw the Y axis..."

| box x start stop increment y |
x ← self graphBox left - self thickness y.
y ← self graphBox bottom + self thickness x.
box ← x @ self graphBox top corner: self graphBox left @ y.
Display fill: box rule: Form over mask: Form black.
increment ← interval y.
start ← self dataWindow bottom.
stop ← self dataWindow top.
(start to: stop by: increment) do: [:value | self displayYTick: value]
```

displayTitle

```
"Display the title..."

| text |
text ← Text string: self title emphasis: self titleEmphasis.
text ← text asDisplayText.
text align: text boundingBox center with: self titleBox center.
text
    displayOn: Display
    at: 0 @ 0
    clippingBox: Display boundingBox
    rule: Form under
    mask: Form black
```

displayView

```
"Display a waveform..."

self resetVariables.
self clear.
self displayTitle.
self displayOrdinate.
self displayAbscissa.
self displayData.
```

displayXTick: value

```
"Display an X tick mark..."

| x y pen text |
x ← (self dataTransformation applyTo: value @ 0) x.
y ← self graphBox bottom + self thickness x.
pen ← Pen new.
pen place: x @ y.
pen down.
pen goto: x @ (y + self length y).
text ← value printString asDisplayText.
text align: text boundingBox topCenter with: x @ (y + self length x).
text
    displayOn: Display
    at: 0 @ 0
    clippingBox: Display boundingBox
    rule: Form under
    mask: Form black
```

displayYTick: value

```
"Display a Y tick mark..."

| box x y start stop increment pen text |
x ← self graphBox left - self thickness y.
y ← (self dataTransformation applyTo: 0 @ value) y.
pen ← Pen new.
pen place: x @ y.
pen down.
pen goto: x - self length y @ y.
text ← value printString asDisplayText.
text align: text boundingBox rightCenter with: x - self length y @ y.
text
    displayOn: Display
    at: 0 @ 0
    clippingBox: Display boundingBox
    rule: Form under
    mask: Form black
```

resetVariables

```
"Calculate new values for some of our variables..."

| box |
box ← self insetDisplayBox.
self graphBox:
    (box
        insetOriginBy: self leftMargin @ self topMargin
        cornerBy: self rightMargin @ self bottomMargin).
self dataBox: (self graphBox insetBy: self graphBox extent * self dataInsets).
self dataTransformation:
    (WindowingTransformation window: self dataWindow viewport: self dataBox)
```

WaveformView methods for: display boxes

The text and title boxes demarcate where the cursor position and title text will be drawn.

WaveformView methods for: display boxes

```
textBox
    "Return the text box..."

    | textBox |
    textBox ← self insetDisplayBox copy.
    textBox top: textBox bottom - (self bottomMargin / 2).
    ^textBox

titleBox
    "Return the title box..."

    | titleBox |
    titleBox ← self insetDisplayBox copy.
    titleBox bottom: self graphBox top.
    ^titleBox
```

WaveformView methods for: text updating

The **cursorPosition** method translates the current sensor position into data relative coordinates.

The **updatePosition:** method draws the current data relative X and Y cursor position in our text box.

The **updateText** method coordinates this update, and ensures that it is only done when the cursor position changes (which makes the display flash less when the cursor is not being moved).

WaveformView methods for: text updating

```
cursorPosition
    "Return the cursor position..."

    | point |
    point ← self controller sensor cursorPoint.
    ^(self dataTransformation applyInverseTo: point) rounded
```

```
updatePosition: position
    "Update the position text..."

    | text |
    Display fill: self textBox mask: self insideColor.
    text ← 'X:  ' , position x printString , '    ' , 'Y:  ' , position y printString.
    text ← text asDisplayText.
    text align: text boundingBox center with: self textBox center.
    text
        displayOn: Display
        at: 0 @ 0
        clippingBox: Display boundingBox
        rule: Form under
        mask: Form black

updateText
    "Update the displays..."

    | position |
    position ← self cursorPosition.
    self lastPosition ~= position
        ifTrue:
            [self updatePosition: position.
            self lastPosition: position]
```

WaveformView methods for: updating

When we get a #*waveform* update request, we clear our view, and redraw the entire display.

WaveformView methods for: updating

```
update: aSymbol
    "Do everything over for now..."

    aSymbol == #waveform
        ifTrue:
            [self clear.
            self displayView]
```

Chapter IV -- Anatomy of the Battery Library

The six categories given herein define the battery simulation's relatively application independent code. These categories are:

Realtime-Support	Simulated realtime timebase code
Realtime-Devices	Clocks, digitizers, digital I/O devices
Waveform-Support	Waveform collection and statistics
Random-Support	Random integer and stream support
Plumbing-Support	Objects for connecting streams together
Accessable-Objects	Support for record/dict. transparency

The Realtime-Support and Realtime-Devices categories together simulate the realtime hardware that laboratory applications must call upon to conduct data acquisition and experimental control.

The Waveform-Support category contains support for multichannel waveform collections, as well as a stream-style averager and statistical collection class.

The Plumbing-Supply category contains a variety of fixtures for connecting stream-objects together. These are used to construct the stream-style averaging and statistical tools mentioned above.

The Accessable-Object category contains a pair of classes that allow any object to in effect add new instance variables dynamically. These new fields are kept in a per-instance dictionary. Attempts to use the conventional instance variable accessing protocols are translated into dictionary references. Attempts to access an AccessableObject using the dictionary accessing protocols will allow access to instance variables as well.

Once again, the code for each system category is presented in turn. A general description of each category is given first, along with a list of the classes defined in that category. Each class in the category is then presented. Each class description begins with a table that resembles a standard Smalltalk class definition template. The class name and its superclass are given, first followed by lists of instance and class variables. Inherited instance variable names are given in *italics*. These lists are followed by the pool dictionary list, the category name, and a table showing the class hierarchy.

Realtime-Support

The Realtime-Support system category contains but one class in the current battery simulation scheme:

Timebase	Generates a simulated timebase

Timebase

The Timebase class provides a simulated timebase for the simulated clocked realtime devices. It is unusual in that all its methods are defined in its metaclass. Timebase objects keep their simulated timebase in a metaclass instance variable named *tick*.   Devices that want to connect themselves with the Timebase object do so by making themselves dependents of Timebase. Timebase ticks themselves are declared when someone sends a **doTick** message to Timebase. Timebase's job then is two-fold: it counts the tick, and declares itself to have changed.

Timebase

class name Timebase
superclass Object
instance variable names
class variable names
pool dictionaries
category Realtime-Support
hierarchy
Object
Timebase
comment
Timebase objects generate a simulated timebase for clocked realtime device objects.

Timebase class

class Timebase class
superclass Object class
instance variables
tick
process

Timebase class methods for: class initialization

The class initialization method zeros the tick count and breaks any previously existing dependent entries that might have existed for us. (One might be surprised how much trouble one can get into if one doesn't do this, what with old versions of various devices on the heap and all.)

Timebase class methods for: class initialization

```
initialize
    "Initialize our instance variables, and rid ourself of old dependents."
    "Timebase initialize"

    tick ← 0.
    self breakDependents
```

Timebase class methods for: instance creation

We override **new** to ensure that no <u>instances</u> of Timebase are created.

Timebase class methods for: instance creation

```
new
    "Do not permit this."

    self error: 'Timebase should have no instances...'
```

Timebase class methods for: accessing

We allow external access to the tick counter.

Timebase class methods for: accessing

```
tick
    "Return the number of ticks we've counted since we were last initialized."

    ^tick

tick: anInteger
    "Explicitly set the tick counter."

    ^tick ← anInteger
```

Timebase class methods for: timebase process

When a tick is declared, we increment the tick count, and declare that a change has occurred. This will send an **update:** message to all the Timebase's dependents, with Timebase as the argument. This protocol's name is a vestige of an earlier implementation of Timebase that used Smalltalk processes. (One early reader of this work, upon seeing the current dependent-based scheme for driving timebase dependent simulated devices remarked that he couldn't imagine anything much slower. The current Timebase scheme is indeed quite slow. However, that reader had never seen an earlier process-based scheme in action.)

Timebase class methods for: timebase process

```
doTick
    "A tick has occurred.  Up our tick count, and say we changed."

    tick ← tick + 1.
    self changed
```

waitForEvent

"Declare a clock tick. This will in turn update any timebase dependent dependents."

```
self doTick
```

Realtime-Devices

The Realtime-Devices system category contains simulations of the sorts of realtime peripheral devices typically used in laboratory application environments. The classes in this category are:

Device	Abstract superclass for realtime devices
ClockedDevice	Superclass for Timebase dependents
Clock	A simulated programmable clock class
Digitizer	Abstract clocked A/D converter
BufferedDigitizer	A/D converter that writes to a buffer
StreamedDigitizer	A/D converter that writes to a stream
InputBit	A simulated digital input bit
OutputBit	A simulated digital output bit

Device

Device is an abstract superclass for all simulated realtime devices. It adds default initialization protocol, and provides a shared random integer generator.

class name Device
superclass Object
instance variable names
class variable names
DeviceAccess
DeviceIntegerGenerator
pool dictionaries
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
This abstract class provides common realtime device behavior...

Device methods for: instance initialization

Device methods for: instance initialization

initialize
"Default for Devices is to do nothing."

^self

Device class

class Device class
superclass Object class

Device class methods for: instance creation

Device class methods for: instance creation

new

"Create a new device, and initialize it. (This ensures that any instance of device is initialized upon creation (as per usual practice).)"

```
| device |
device ← super new.
device initialize.
^device
```

Device class methods for: class initialization

Device class methods for: class initialization

initialize

"Device initialize"
"Create a shared random number generator and a semaphore. (The semaphore is not used by the current synchronous realtime device implementation.)"

```
DeviceAccess ← Semaphore forMutualExclusion.
DeviceIntegerGenerator ← IntegerGenerator new
```

Device initialize

ClockedDevice

Clocked devices are an abstract superclass for all the simulated realtime devices that require that they receive Timebase updates. (All the simulated devices in the current battery simulation except for OutputBit fit this description.)

class name ClockedDevice
superclass Device
instance variable names
class variable names
DeviceAccess
DeviceIntegerGenerator
pool dictionaries
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
ClockedDevice is an abstract class that adds protocol for setting up Timebase dependencies to Device.

ClockedDevice methods for: instance initialization

Each new clocked device is a dependent of the Timebase object.

ClockedDevice methods for: instance initialization

```
initialize
    "All ClockedDevice instances should be made Timebase dependents."

    super initialize.
    Timebase addDependent: self
```

ClockedDevice methods for: timebase access

The default behavior for a ClockedDevice when the Timebase declares an update is to tell itself that a tick has occurred by sending **doTick** to itself.

ClockedDevice methods for: timebase access

```
update: anObject
    "Our timebase must have issued an update request.
    Do what we need to do when a tick occurs.
    (We keep this method in reserve in case additional update
    protocol needs to be added somewhere.)"

    self doTick
```

Clock

The Clock class simulates a programmable realtime clock. These clocks are based on the six hardware and four software timers found in the Pearl II LABPAK library (See [Heffley 85] and [Foote 85]).

Class Clock allows the user to start a clock for an indicated interval (in Timebase ticks), and test for whether this interval has elapsed. While a clock is running, the number of ticks since it was started can be ascertained.

A Clock may be run either in *single* or *repeat* mode. In single mode, the clock is stopped when the interval originally requested has elapsed. In repeat mode, the Clock is restarted from zero when the original interval elapses. A particularly useful feature of these timers is that a block may be scheduled to run either each time a timer completes an interval, or at any timepoint during a timer's interval. (This feature, in fact, might serve as the basis for a simulated parallel process-based stimulus generation and response reporting scheme.)

class name Clock
superclass ClockedDevice
instance variable names
interval
active
count
flag
cycle
recycle
scheduledBlocks
startTick
class variable names
DeviceAccess
DeviceIntegerGenerator
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
Clock is a realtime programmable clock simulation. It provides protocol

for starting and stopping timers in a number of different modes...

Clock methods for: instance initialization

We make sure each new Clock is marked as off when it is created.

Clock methods for: instance initialization

```
initialize
    "Make sure the active flag has a value, then let super make the
    dependent list connections."

    active ← false.
    flag ← false.
    super initialize
```

Clock methods for: clock control

These messages define the basic clock control protocol. The **atTime:do:** message allows a given block to be scheduled for execution when the clock reaches the given timepoint.

The mechanism for keeping track of this is somewhat interesting. Each instance of Clock keeps a Dictionary in its *scheduledBlocks* instance variable. When a block for a given timepoint is scheduled, we check to see if there is an entry in this dictionary keyed by the given timepoint. If such an entry exists, we add the block given us to the OrderedCollection of blocks for that timepoint. If there is no entry for the given key, we create one and place a new OrderedCollection with the given block in it.

The **startEvery**: method starts a Clock in repeat mode using the given interval and no completion block.

The **startEvery:thenDo:** method starts a clock in repeat mode and schedules the given block to be executed <u>every</u> <u>time</u> the clock times out.

The **startFor:** method starts a clock in single mode using the given interval and no completion block.

The **startFor:thenDo:** method starts a clock in single mode and schedules the given block for execution when this interval has elapsed. Each of these methods calls the private method **startInterval:recycle:thenDo:** to actually manipulate the Clock.

The **stop** message may be called at any time to turn off a given Clock.

Clock methods for: clock control

```
atTime: tick do: aBlock
    "Add this block to the OrderedCollection for the given time point.  If
    there is no collection in the Dictionary for this time point, make one."

    (scheduledBlocks at: tick ifAbsent:
        [scheduledBlocks at: tick put: (OrderedCollection new: 4)])
            add: aBlock

startEvery: ticks
    "Run us in repeat mode."

    self
        startInterval: ticks
        recycle: true
        thenDo: nil

startEvery: ticks thenDo: aBlock
    "Run the indicated block at the indicated interval."

    self
        startInterval: ticks
        recycle: false
        thenDo: aBlock

startFor: ticks
    "Start us for the indicated interval."

    self
        startInterval: ticks
        recycle: false
        thenDo: nil

startFor: ticks thenDo: aBlock
    "Start a clock for the indicated interval, and schedule the indicated
    block when it is done."

    self
        startInterval: ticks
        recycle: false
        thenDo: aBlock
```

stop

"Just mark the clock as inactive."

```
active ← false
```

Clock methods for: public access

The public access protocol provides access to information about the state of this Clock. Some values are read directly from instance variables, others are computed. Compare, for example the methods for **elapsed** and **left**.

The **wait** method waits until a Clock times out. It accomplishes this by repeatedly waiting for Timebase events until the one that finally causes the Clock to time out occurs. Similarly, the **waitUntil:** method allows the user to wait until a given Clock reaches a particular timepoint.

Clock methods for: public access

```
elapsed
    "Return how many ticks we've counted."

    ^count

flag
    "Allow the user to test the done flag."

    ^flag

flag: aBoolean
    "Allow the user to set the done flag."

    ^flag ← aBoolean

left
    "Return the amount of time left until the clock times out."

    ^interval - count

wait
    "Just wait for the done flag."

    [flag]
        whileFalse: [Timebase waitForEvent]

waitUntil: tick
    "Just wait 'til we've counted up to tick."

    [count ~= tick]
        whileTrue: [Timebase waitForEvent]
```

Clock methods for: timebase access

The methods in this protocol category define what happens when a Clock receives an update request from the Timebase object. Recall that the **update:** method in ClockedDevice calls **doTick**.

When a Clock receives a **doTick** message, it first check the *active* flag. If this flag is turned off, it ignore this tick. Otherwise, it updates the internal counter, and check to see if any blocks have been scheduled for execution for the current timepoint. If so, it passes the collection in the *scheduledBlocks* dictionary for this timepoint to the **doBlocks:** method. This method executes each block in the collection given it. The Clock then check to see whether the current interval has elapsed. If so, it calls **doTimeout.**

```
Clock methods for:  timebase access

    doBlocks: blocks
        "Run all the blocks in the collection given us.  (Note that we may want
        to think about forking these instead of running them here and now.)"

        blocks do: [:aBlock | aBlock value]

    doTick
        "A timebase tick occurred.  If we are active, up our count, check for scheduled
    blocks,
        and see if we timed out."

        active
            ifTrue:
                [count ← count + 1.
                (scheduledBlocks includesKey: count)
                    ifTrue: [self doBlocks: (scheduledBlocks at: count)].
                count = interval ifTrue: [self doTimeout]]

    doTimeout
        "If this is repeat mode, start over, otherwise just stop."

        recycle
            ifTrue: [count ← 0]
            ifFalse: [active ← false].
        cycle ← cycle + 1.
        flag ← true
```

Clock methods for: private

The **startInterval:recycle:thenDo:** method is called by several of the public clock control methods to turn on a Clock. It accepts an interval, a single/repeat mode flag, and a block to execute when the clock has timed out.

Clock methods for: private

```
startInterval: ticks recycle: aBoolean thenDo: aBlock
    "Initialize the clock variables, and start us up."

    count ← 0.
    interval ← ticks.
    active ← true.
    cycle ← 0.
    recycle ← aBoolean.
    flag ← false.
    startTick ← Timebase tick.
    scheduledBlocks ← (Dictionary new: 4).
    (aBlock~~nil) ifTrue: [self atTime: ticks do: aBlock]
```

Clock class

class Clock class
superclass ClockedDevice class

Clock class methods for: examples

The example below illustrates how several features of Clock are used. The critical section demonstration was a bit more realistic back when this simulation used Smalltalk processes to implement the Timebase.

Clock class methods for: examples

```
example
    "Clock example"
    "MessageTally spyOn: [Clock example]"
    "Time millisecondsToRun: [Clock example]"

    | aClock |

"Set up the timebase..."
    Timebase initialize.

"Create a clock, and schedule a bunch of completion blocks..."
    aClock ← Clock new.
    aClock startFor: 5 thenDo: [Transcript show: 'All done...'; cr].
    aClock atTime: 5 do:
        [Transcript show: 'Time:  ' , aClock elapsed printString; cr].
    aClock atTime: 4 do:
        [Transcript show: 'Time:  ' , aClock elapsed printString; cr].

"This should make things interesting (and demonstrate a classic critical section
problem)..."
    Transcript show: 'During set up:  ' , aClock elapsed printString; cr.
    aClock atTime: 3 do:
        [Transcript show: 'Time:  ' , aClock elapsed printString; cr].
    aClock atTime: 2 do:
        [Transcript show: 'Time:  ' , aClock elapsed printString; cr].
    aClock atTime: 1 do:
        [Transcript show: 'Time:  ' , aClock elapsed printString; cr].
    aClock wait.
    Transcript show: 'After wait...'; cr.
```

Digitizer

These classes simulate clock driven multichannel analog-to-digital (A/D) subsystems. Digitizer is an abstract superclass for two concrete digitizers, BufferedDigitizer and StreamedDigitizer. Most of the code for these two types of digitizers is contained in this class.

class name Digitizer
superclass ClockedDevice
instance variable names
interval
active
count
flag
channels
points
block
point
class variable names
DeviceAccess
DeviceIntegerGenerator

MinValue
MaxValue
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
This abstract class simulates clocked, multichannel A/D converter systems. Subclasses demonstrate different approaches to handling digitizer output data...

Digitizer methods for: instance initialization

Digitizer methods for: instance initialization

initialize
"Make sure the active flag has a value, then let super make the dependent list connections."

```
active ← false.
flag ← false.
super initialize
```

Digitizer methods for: data collection

More data collection methods are defined by subclasses of Digitizer.

Digitizer methods for: data collection

```
stop
    "Just mark the clock as inactive."

    active ← false
```

Digitizer methods for: public access

Digitizer methods for: public access

```
flag
    "Allow the user to test the done flag."

    ^flag

flag: aBoolean
    "Allow the user to set the done flag."

    ^flag ← aBoolean.

wait
    "Just wait for the done flag."

    [flag]
        whileFalse: [Timebase waitForEvent]
```

Digitizer methods for: timebase access

If we have timed out, execute any completion blocks that may be pending, turn off our active flag, and set our done flag. Per timepoint activity is defined by our concrete subclasses.

The **simulatePoint** method uses the three class variables (the IntegerGenerator we inherited from Device, and the own value bounds) to fabricate random A/D-like values.

Digitizer methods for: timebase access

doTimeout

"We are done. Execute our block, turn off the active flag, and say we are done."

```
block ~~ nil ifTrue: [block value].
active ← false.
flag ← true
```

simulatePoint

"Fabricate a point. We might want to make this more interesting later."

```
^DeviceIntegerGenerator from: MinValue to: MaxValue
```

Digitizer class

class Digitizer class
superclass ClockedDevice class

Digitizer class methods for: class initialization

Digitizer class methods for: class initialization

initialize

"Digitizer initialize"
"Define constants for our minimum and maximum values. The values chosen here are consistent with what a real 12 bit signed A/D system would report."

```
MinValue ← -2048.
MaxValue ← 2047
```

Digitizer initialize

BufferedDigitizer

BufferedDigitizers are fairly faithful simulations of the A/D subsystems that exist on the CPL Pearl II systems [Heffley 85]. They allow a caller to stipulate that a given number of A/D channels be sampled repeatedly at a given interval until a given buffer has been filled.

class name BufferedDigitizer
superclass Digitizer
instance variable names
interval
active
count
flag
channels
points
block
point

buffer
class variable names
DeviceAccess
DeviceIntegerGenerator

MinValue
MaxValue
pool dictionaries
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

BufferedDigitizer methods for: data collection

The method below is used to start a BufferedDigitizer. Most of the parameters are explained in the method comment. A user supplied block may optionally be scheduled for execution when a multi-point sweep has completed.

BufferedDigitizer methods for: data collection

collectChannels: chans points: pnts in: buf every: ticks thenDo: aBlock

```
"Collect data from the indicated number of channels at the indicated
interval until the given number of time points have been sampled.  The
data will be stored in a caller supplied buffer.  This buffer must be an
ArrayedCollection with (at least) 'chans' elements.  Each of these
elements must be an ArrayedCollection large enough to accommodated at
least 'pnts' data points."

interval ← ticks.
count ← 0.
active ← true.
flag ← false.
channels ← chans.
point ← 0.
points ← pnts.
buffer ← buf.
block ← aBlock
```

BufferedDigitizer methods for: timebase access

If a BufferedDigitizer is active and a timebase tick occurs, it increments the tick count and sees if the sampling interval has elapsed. If so, it resets the counter and fake some data. For each waveform in the buffer, it adds a simulated point at the offset for the current timepoint. When it has filled all the points in the buffer, it declares a timeout.

BufferedDigitizer methods for: timebase access

doTick

```
"Fake some data."

active
    ifTrue:
        [count ← count + 1.
        count = interval ifTrue:
        [count ← 0.
        point ← point + 1.
        buffer do: [:waveform | waveform at: point put: self simulatePoint].
        point = points ifTrue: [self doTimeout]]]
```

"Brian Foote 7/21/87 (?!) Repaired to fix missing interval timeout code..."

StreamedDigitizer

StreamedDigitizers are much like BufferedDigitizers, except that instead of filling a buffer with data, they write them down an output stream of some sort as they are collected. This capability, together with the Plumbing-Support classes for constructing dataflow fixtures, might make an interesting combination, especially if support for piping data between processes were implemented. This implementation of the battery does not use StreamedDigitizers.

class name StreamedDigitizer
superclass Digitizer
instance variable names
interval
active
count
flag
channels
points
block
point

stream
class variable names
DeviceAccess
DeviceIntegerGenerator

MinValue
MaxValue
pool dictionaries
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

StreamedDigitizer methods for: data collection

The data collection start up protocol for StreamedDigitizers is much like that for BufferedDigitizers, except that a WriteStream of some is given (in place of the buffer).

StreamedDigitizer methods for: data collection

```
collectChannels: chans points: pnts on: aStream every: ticks thenDo: aBlock
    "Collect data from the indicated number of channels at the indicated interval until the
given    number of time points have been sampled.  The data will be written to a caller
supplied stream..."

    interval ← ticks.
    count ← 0.
    active ← true.
    flag ← false.
    channels ← chans.
    point ← 0.
    points ← pnts.
    stream ← aStream.
    block ← aBlock
```

StreamedDigitizer methods for: timebase access

This method looks pretty much like the one in BufferedDigitizer, except that instead of filling a buffer, it sends points down the output stream every time an inter-point interval elapses.

StreamedDigitizer methods for: timebase access

```
doTick
    "Fake some data..."

    active
        ifTrue:
            [count ← count + 1.
            count = interval ifTrue:
            [count ← 0.
            point ← point + 1.
            (1 to: channels) do: [stream nextPut: self fakePoint].
            point = points ifTrue: [self doTimeout]]]

"Brian Foote  7/21/87  Added some missing interval timeout code..."
```

InputBit

Class InputBit simulates a set of hardware and software capabilities that allow a digital input device to serve as a parallel source of time stamped events. In this simulation, each line of such an parallel point is simulated as a separate instance of InputBit. A line number is given when an InputBit is created. The simulation does not check for whether a given line is used for more than one purpose at a time. In practice, of course, such misuse could cause problems.

class name InputBit
superclass ClockedDevice
instance variable names
active
count
flag
bit
clock
time
block
fakeTime
low
high
class variable names
DeviceAccess
DeviceIntegerGenerator
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
This class simulates a single line parallel input device.

InputBit methods for: instance initialization

InputBit methods for: instance initialization

initialize
"Set out reasonable defaults."

active ← false.

```
flag ← false.
super initialize.
```

InputBit methods for: input reporting

Before an InputBit can report events, it must be enabled. The three methods defined here allow a bit to be merely enabled, enabled in association with a given Clock, or enabled with a Clock and a block that will be executed whenever an input event occurs for the given bit.

The **disable** method disarms a previously armed InputBit.

InputBit methods for: input reporting

```
disable
    "Mark us as off."

    active ← false

enable
    "Turn us on with no clock or block."

    self enableUsingClock: nil onInputDo: nil

enableUsingClock: aClock
    "Turn us on, saving the given clock."

    self enableUsingClock: aClock onInputDo: nil

enableUsingClock: aClock onInputDo: aBlock
    "Turn us on, saving the given clock and block."

    clock ← aClock.
    block ← aBlock.
    flag ← false.
    count ← 0.
    active ← true
```

InputBit methods for: accessing

The **bit** and **bit**: methods allow access to a simulated line number. The **fakeTimeLow:high:** method can be called by the user to indicate that an event be faked (forced) for this InputBit at a time randomly selected within the time range given.

The **time** method returns the time at which the last input event occurred for this InputBit, relative to the associated Clock (if any). If no event has occurred, or no Clock is associated with this bit, the **time** method returns *nil*.

InputBit methods for: accessing

```
bit
    "Return the bit that I watch."

    ^bit

bit: bitNumber
    "Set the bit number."

    ^bit ← bitNumber

fakeTimeLow: lowLimit high: highLimit
    "Turn on faking."

    low ← lowLimit.
    high ← highLimit.
    fakeTime ← DeviceIntegerGenerator integerBetween: low and: high

time
    "Return the saved time or nil."

    ^time
```

InputBit methods for: timebase access

InputBits handle Timebase ticks by first checking to see whether they are active. If not, the tick is ignored. If the bit is active, the InputBit increments the tick count, and checks to see if any mouse button is currently pressed. If so, or if a forced fake input time has is due for this bit, it will declare an input event.

The **doEvent** method is called when an input event occurs. It sees if a block must be executed, records the input time relative to the associated Clock (if any), sets the event flag, and turns off the bit. Note that because the first detected event disables an InputBit, programs that expect multiple events must take care to quickly detect and rearm their input bits.

InputBit methods for: timebase access

doEvent

"An input event for this bit has been generated (somehow). Set the event flag to true and the active flag to false. Then, record the event time relative to the given clock, if any. Then run the completion block, if need be. Note that all this has the effect of latching the first event time after a bit is enabled. Subsequent events will be ignored until the bit is reenabled."

```
flag ← true.
active ← false.
clock ~~ nil ifTrue: [time ← clock elapsed].
block ~~ nil ifTrue: [block value]
```

"Notes 4/29/86: It might be a good idea to make the order in which these tasks are done consistent across devices. As long as we use a synchronous event base this won't matter that much..."
"Brian Foote 7/21/87 Changed the order of these actions so that completion blocks could reenable the bits, if desired..."

doTick

"Count ticks, and check for mouse buttons and faked events"

```
active
    ifTrue:
        [count ← count + 1.
        Sensor
            anyButtonPressed |
            (fakeTime ~~ nil and: [count = fakeTime])
                ifTrue: [self doEvent]]
```

InputBit methods for: public access

InputBit methods for: public access

flag

"Allow the user to test the done flag."

```
^flag
```

flag: aBoolean

"Allow the user to set the done flag."

```
^flag ← aBoolean
```

wait

"Just wait for the done flag."

```
[flag]
    whileFalse: [Timebase waitForEvent]
```

InputBit class

class InputBit class
superclass ClockedDevice class

InputBit class methods for: instance creation

InputBit class methods for: instance creation

new

```
"Force a bit number..."

self shouldNotImplement
```

onBit: anInteger

```
"Save the bit number..."

| newBit |
newBit ← super new.
newBit bit: anInteger.
^newBit
```

InputBit class methods for: examples

InputBit class methods for: examples

example

```
    "InputBit example"
    "MessageTally spyOn: [InputBit example]"
    "Time millisecondsToRun: [InputBit example]"

    | aBit aClock |

"Start the timebase..."
    Device initialize.
    Timebase initialize.

"Set up a bit and a clock..."
    aClock ← Clock new.
    aClock startFor: 1000.
    aBit ← InputBit newOnBit: 0.
    aBit fakeTimeLow: 200 high: 800.
    aBit enableUsingClock: aClock.
```

```
"Wait for the bit event, and print the time..."
    aBit wait.
    Transcript show: 'Time was:  ' , aBit time printString ; cr.
```

OutputBit

The OutputBit class simulates a parallel digital output system. Each line of the simulated output port is represented by an instance of Output bit.

class name OutputBit
superclass Device
instance variable names
bit
class variable names
DeviceAccess
DeviceIntegerGenerator
pool dictionaries
category Realtime-Devices
hierarchy

```
Object
    Device
        ClockedDevice
            Clock
            Digitizer
                BufferedDigitizer
                StreamedDigitizer
            InputBit
    OutputBit
```

comment
This class simulates a single line parallel output device.

OutputBit methods for: accessing

OutputBit methods for: accessing

bit
"Return the bit number..."

^bit

bit: bitNumber
"Set the bit number..."

^bit ← bitNumber

OutputBit methods for: set/clear

Note that the **set** and **clear** methods do nothing what-so-ever. The buck truly stops here.

OutputBit methods for: set/clear

clear

"This is only a simulation..."

```
^self
```

set

"The buck stops here..."

```
^self
```

OutputBit class

class	OutputBit class
superclass	Device class

OutputBit class methods for: instance creation

OutputBit class methods for: instance creation

new

"Force a bit number..."

```
self shouldNotImplement
```

onBit: anInteger

"Save the bit number..."

```
| newBit |
newBit ← super new.
newBit bit: anInteger.
^newBit
```

Waveform-Support

The Waveform-Support system category contains classes that manage waveform data and statistical information. The classes in this category are:

Averager	A stream-style incremental averager
Waveform	A collection of data
WaveformCollection	A collection of waveforms
Tally	A streamed incremental statistics collector

Averager

Averager objects provide a stream-oriented framework for incrementally averaging anything that adheres to a given set of protocols. An instance of Averager is given a sum object when it is created into which individual observations are accumulated. Individual observations are fed to an Averager using the standard WriteStream message **nextPut:**.

Sum objects must be able to respond to the messages **+=**, **/=**, and **zero**. Objects given to an Averager by **nextPut:** must be suitable arguments to the **+=** method of the sum object. Averager objects take care of counting the number of observations they have seen, and can be asked to report a current average at any time.

class name Averager
superclass Object
instance variable names
    count
    sum
class variable names
pool dictionaries
category Waveform-Support
hierarchy
    Object
        Averager

Averager methods for: instance initialization

The **initialize** method merely makes sure that the counter is zeroed when we begin.

Averager methods for: instance initialization

```
initialize
    "Make sure new averages start with their counters at zero..."

    self count: 0
```

Averager methods for: accessing

The accessing methods allow the current observation count to be easily ascertained, and could be used to insert a previously computed set of sums and counts should it be necessary to pick up a previous average where it had left off.

Averager methods for: accessing

```
count
    "Return the number of objects incorporated into our sum buffer so far..."

    ^count

count: n
    "Set the counter as indicated.  This might be useful were one to want to
    add new data into a saved pair of sum and count objects..."

    ^count ← n

sum
    "Return the current sum object.  (This can be any object that responds to
    '+=', '/=', and 'zero'.)"

    ^sum

sum: anObject
    "Set the current sum object.  (This can be any object that responds to
    '+=', '/=', and 'zero'.)"

    ^sum ← anObject
```

Averager methods for: averaging

This category defines the primary external averaging protocol. An Averager can be reinitialized using the **zero** method. Individual observations are reported to us using the **nextPut:** method.

The use of a steam-oriented protocol allows Averagers to be easily incorporated into a pipelined dataflow scheme of the sort made possible by the classes in the Plumbing-Support system category.

The **nextPut:** method sends the object it receives to the sum object as part of an add-to-self message. The **+=** selector for this message is modelled after a C [Kernighan 78] operator that performs an analogous function. For examples of how averagable objects might implement these messages, see classes Waveform and WaveformCollection.

The **average** method similarly employs a C-style **/=** operator to divide the sum object by the accumulated count. The use of the **+=** and **/=** operators represents an attempt to take into account the the relative runtime efficiency differences between, for example, adding one collection to another vs. creating a new collection, adding two collections into it, and discarding one of the old ones. Such considerations are important when realtime constraints come into play. One can, however, make a case that using **+** and **/** as the averagable object protocol might have given the Averager greater potential generality.

Averager methods for: averaging

```
average
^self sum /= self count

nextPut: anObject
    "Tell our sum object to add this object into to itself.  Then update our
    count... "

    self sum += anObject.
    self count: self count + 1

zero
    "Forward this to our sum object, if  any..."

    count ← 0.
    sum isNil ifFalse: [sum zero]
```

Averager class

class Averager class
superclass Object class

Averager class methods for: instance creation

Averagers can either be created with a designated summing object, or created without one with the expectation that one will be assigned later.

Averager class methods for: instance creation

new

```
"Create a new averager..."

| avg |
avg ← super new.
avg initialize.
^avg
```

on: anObject

```
"Create a new averager and set its initial sum buffer as indicated. (This can
be any object that responds to '+=', '/=', and 'zero'.)"

| avg |
avg ← self new.
avg sum: anObject.
avg zero.
^avg
```

Waveform

Waveform objects simulate digitized time-series waveforms. They add protocol for simulating A/D data, and for conforming to the Averager's averagable object protocol to Array.

class name Waveform
superclass Array
instance variable names
class variable names
pool dictionaries
category Waveform-Support
hierarchy

```
Object
    Collection
        SequenceableCollection
            ArrayedCollection
                Array
                    Waveform
```

comment
Waveforms represent single channels time series records...

Waveform methods for: instance initialization

The **initialize** method makes sure that all Waveforms are initially filled with zeros. Since a **zero** method is required to conform to the protocol required by the Averager, Waveform uses this method to set all the Waveform's elements to zero.

Waveform methods for: instance initialization

```
initialize
    "Let's zero new waveforms..."

    self zero
```

Waveform methods for: element initialization

Waveforms know how to do two things to all their elements. One is to set them all to zero. The other is to fill each element with random integer values of the sort that might be returned by an A/D system. It is easy to imagine other useful capabilities that might be added here, such as sinusoid, square, ramp, and triangle wave creation.

Waveform methods for: element initialization

simulateData

```
"Put a random number in the range that one would expect from an A/D
converter into each or our elements..."

| r |
r ← IntegerGenerator new.
(1 to: self size)
    do: [:p | self at: p put: (r from: -2048 to: 2047)]
```

"Notes 4/30/86: We might want to consider creating a pool with A/D ranges and the like in it..."

zero

```
"Filling a collection with zeros is easy..."

self atAllPut: 0
```

Waveform methods for: averaging

Two C-style averaging methods required by the Averager are implemented here. The **+=** method allows us to do an element-wise addition of all the elements of a given waveform into ourself. The **/=** method divides each of the Waveform's elements by the given value, and rounds the result to the nearest integer.

Waveform methods for: averaging

+= aWaveform

```
(1 to: self size)
    do: [:p | self at: p put: (self at: p)
                + (aWaveform at: p)]
```

/= count

```
| w |
w ← self collect: [:pnt | pnt / count rounded].
^w
```

Waveform methods for: extrema

These methods are called primarily by WaveformView. (Waveform could do worse than use them internally.)

Waveform methods for: extrema

```
highX
    ^self size

highY
    ^2047

lowX
    ^1

lowY
    ^-2048
```

Waveform class

class	Waveform class
superclass	Array

Waveform class methods for: instance creation

The **points:** instance creation method gives us a pithy mechanism for creating new Waveform objects.

Waveform class methods for: instance creation

```
points: size
    "Create a waveform of the given size..."

    | w |
    w ← super new: size.
    w initialize.
    ^w
```

WaveformCollection

WaveformCollections are (as the name suggests) collections of Waveform object. They implement much of the same external protocol as does Waveform, so that operations such as the averaging protocol and data simulation can be done en-masse on several waveforms at one.

WaveformCollections model the two dimensional arrays that a conventional programming language might employ to store multi-channel waveform data.

class name WaveformCollection
superclass Array
instance variable names
class variable names
pool dictionaries
category Waveform-Support
hierarchy

```
Object
    Collection
        SequenceableCollection
            ArrayedCollection
                Array
                    WaveformCollection
```

WaveformCollection methods for: element initialization

These element initialization methods simply forward their respective selectors to each of the WaveformCollection's constituent Waveform objects.

WaveformCollection methods for: element initialization

simulateData

```
"Tell all our elements to fill themselves with bogus data..."

self do: [:chan | self at: chan simulateData]
```

zero

```
"Tell all our elements to zero themselves..."

self do: [:chan | chan zero]
```

WaveformCollection methods for: averaging

WaveformCollections implement the averaging protocol by applying the averaging selectors in turn to each of their components.

WaveformCollection methods for: averaging

+= aWaveformCollection

```
"Sum each of the elements of the given waveform collection into the
respective element of us..."

self with: aWaveformCollection do:
    [:selfElement :otherElement | selfElement += otherElement]
```

/= count

```
"Divide each of our elements by the given value..."

self do:
    [:waveform | waveform /= count]
```

WaveformCollection class

class	WaveformCollection class
superclass	Array class

WaveformCollection class methods for: instance creation

The instance creation protocol provides a pithy means of creating a WaveformCollection with the indicated number of channels, each of which will be a Waveform with the indicated number of points.

WaveformCollection class methods for: instance creation

channels: chans points: pnts

```
"Create a waveform collection with 'size' elements, each of which is a
waveform..."

| w |
w ← super new: chans.
(1 to:  chans)
    do: [:elem | w at: elem put: (Waveform points: pnts)].
^w
```

new

```
"We need to know dimensions to make a waveform collection..."

self shouldNotImplement
```

Tally

Tally objects are WriteStream-like objects that collect statistical information about the data that is written into them. The data kept are: a count of the number of observations written into the Tally object, the running sum and sum of the squares of the data, the average for the data, and the variance and standard deviation of the data.

Tally objects were designed to be used with the streamed dataflow fixture classes defined in the Plumbing-Support category. See the plumbing fixture set up code in BatteryItem for an example of their use. A major benefit of encapsulating the strategies for statistical data collection in objects like Tally objects is that by extending or subclassing objects like these, one can easily add code for additional statistics and have it be available for every data collection category for which Tally-like fixtures are used.

class name Tally
superclass Object
instance variable names
  sum
  sumSquared
  count
class variable names
pool dictionaries
category Waveform-Support
hierarchy
  Object
    Tally

Tally methods for: instance initialization

The collection of standard deviations requires that the Tally object keep a count, a running sum and a running sum of squares. All of the other summary statistics are a special case of these. New Tally objects begin life with all their accumulating variables zeroed.

Tally methods for: instance initialization

initialize
"Tell ourself to zero our accumulators..."

self zero

```
zero
    "Zero our counter, and our two sums..."

    count ← 0.
    sum ← 0.
    sumSquared ← 0
```

Tally methods for: statistics

Tally objects calculate averages and standard deviations in the conventional fashions (after first guarding for counts of zero). As for the variance, the best one can say is that this is a good example of how object-oriented programming allows us to prototype a function like this and substitute a more efficient algorithm at a later date.

Tally methods for: statistics

```
average
    "Return the average value across all the values we've seen..."

    count == 0
        ifTrue: [^0]
        ifFalse: [^sum / count]

standardDeviation
    "Return the accumulated standard deviation..."

    | t |
    count == 0 ifTrue: [^0].
    t ← count * sumSquared - (sum * sum).
    ^t sqrt / count

variance
    "There are certainly more efficient ways to do this one..."

    ^self standardDeviation * self standardDeviation
```

Tally methods for: accessing

Tally objects make a virtue of necessity by making the count, running sum and sum of squares counters available to the public.

Tally methods for: accessing

```
count
    "Return our current counter value..."
```

```
    ^count

count: n
    "Set the counter value..."

    ^count ← n

sum
    "Return the current running sum..."

    ^sum

sum: n
    "Set the sum accumulator..."

    ^sum ← n

sumSquared
    "Return the squares sum..."

    ^sumSquared

sumSquared: n
    "Set the squares sum..."

    ^sumSquared ← n
```

Tally methods for: updating

Tally objects use the WriteStream convention of using **nextPut:** to write data to Tally objects. This convention allows Tally objects to be used with the plumbing fixture classes, which allow arbitrary stream-like objects to be connected together. (The comment in **nextPut:** notes that earlier versions of Tally used **addIn:** or **+=** for this function.)

By using the standard protocol for streams to implement these data accumulation objects, the streamed digitizers, and the plumbing support objects, the potential generality of all is increased. Standard protocols are discussed in a paper by Johnson [Johnson 88].

Tally methods for: updating

```
nextPut: value
    "Sum the given value into our accumulators. (We used to call this addIn: and +=)..."
```

```
sum ← sum + value.
sumSquared ← sumSquared + (value * value).
count ← count + 1
```

Tally class

class Tally class
superclass Object class

Tally class methods for: instance creation

Tally class methods for: instance creation

new

```
"Create a tally, and initialize it..."

| aTally |
aTally ← super new.
aTally initialize.
^aTally
```

Random-Support

The Random-Support system category contains a set of classes that aid in the production of sequences of random integers and other objects. These classes are:

IntegerGenerator	Returns an integer in a given interval
IntegerStream	Returns integers in a given range
RandomStream	Samples a collection with replacement
SampledStream	Samples without replacement

IntegerGenerator

IntegerGenerator objects are subclasses of Random that return an integer in a given range. (This class has been renamed RandomInteger in newer versions of the battery simulation.)

class name IntegerGenerator
superclass Random
instance variable names
seed
class variable names
pool dictionaries
category Random-Support
hierarchy

```
Object
    Stream
        Random
            IntegerGenerator
                IntegerStream
```

comment
IntegerGenerators return random integers within given ranges...

IntegerGenerator methods for: accessing

The **from:to:** method returns an integer in the given range. Either of the extrema may be returned (that is, the range is inclusive).

IntegerGenerator methods for: accessing

```
from: start to: stop
    "Return a random integer in the range from start to stop inclusive."

    | x range |
    range ← stop - start + 1.
    x ← super next.
    ^(x * range) truncated + start
```

IntegerStream

An IntegerStream is an infinite stream of integers randomly chosen from within a designated closed interval. This interval is established when an IntegerStream object is created, and may be modified by the user.

class name IntegerStream
superclass IntegerGenerator
instance variable names
seed

start
stop
class variable names
pool dictionaries
category Random-Support
hierarchy

```
Object
    Stream
        Random
            IntegerGenerator
                IntegerStream
```

IntegerStream methods for: accessing

The **next** method sends a **from:to:** to IntegerStream's superclass to create a random integer in the range stipulated by the endpoints of the interval. Note that we allow explicit read/write access to these values here.

IntegerStream methods for: accessing

```
next
    "Return a random integer in the range from start to stop inclusive."

    ^self from: start to: stop

start
    "Return our starting (lower) bounding value."

    ^start

start: anInteger
    "Set our start bound."

    ^start ← anInteger

stop
```

```
    "Return our stop bound."

    ^stop

stop: anInteger
    "Set our stop bound."

    ^stop ← anInteger
```

IntegerStream class

class IntegerStream class
superclass IntegerGenerator class

IntegerStream class methods for: instance creation

The **from:to:** instance creation method creates a read stream that will generate uniformly distributed random integers from within the given closed interval.

IntegerStream class methods for: instance creation

```
from: startInteger to: stopInteger
    "Create a new IntegerStream with the specified bounds."

    | temp |
    temp ← self new.
    temp start: startInteger.
    temp stop: stopInteger.
    ^temp
```

RandomStream

RandomStreams are ReadStream-like objects. They are created **on:** a given collection much as a ReadStream is. The difference is that each time an element is requested from a RandomStream, an element of the base collection is selected at random and returned. Sampling is with replacement. Hence, RandomStreams never run out of elements.

class name RandomStream
superclass Stream
instance variable names
    collection
    rand
class variable names
pool dictionaries
category Random-Support
hierarchy

```
Object
    Stream
        RandomStream
            SampledStream
```

comment
RandomStreams return randomly chosen elements (with replacement) from the collection given them. They never run out. (That is, atEnd will always be false.)

RandomStream methods for: accessing

RandomStreams support the essential portions of the Smalltalk ReadStream protocol conventions.

The **atEnd** method is implemented to always return false, because since we are sampling with replacement, we will never run out of elements.

The **contents** method is implemented so that it returns a collection of the same size as the original base collection with elements randomly chosen (with replacement) from it. Also, **size** returns the size of the base collection, even though **next** will return elements forever.

RandomStream methods for: accessing

```
atEnd
    "We'll allow random elements to be chosen forever."
```

^false

collection
"Return our collection."

^collection

collection: aCollection
"Set our collection."

^collection ← aCollection

contents
"Return elements of the base collection in a random order. Note that the resulting collection will be of the same size as the base collection, and that some elements may appear multiple times, while others do not appear at all. Note that a scrambled stream containing all elements of the base collection can be had by defining a SampledStream over it..."

```
| temp |
temp ← collection copy.
^temp collect: [:each | self next]
```

next
"Pick an element at random and return it."

^collection at: (rand from: 1 to: self size).

nextPut: anObject
"We can't do this."

self shouldNotImplement

rand
"Return our IntegerGenerator."

^rand

rand: anIntegerGenerator
"Set our IntegerGenerator."

^rand ← anIntegerGenerator

size
"Size is whatever the size of our collection is."

^collection size

RandomStream class

class RandomStream class
superclass Stream class

RandomStream class methods for: instance creation

The instance creation messages allow an instance of RandomStream to be created over either all or a portion of any collection that can be coerced into an OrderedCollection. Note that we work from a copy of the given collection rather than the original. This becomes important in subclasses of RandomStream like SampledStream that alter their copies of the base collection.

RandomStream class methods for: instance creation

```
on: aCollection
    "Create ourself, and set up a IntegerGenerator..."

    | temp |
    temp ← self basicNew.
    temp collection: aCollection copy asOrderedCollection.
    temp rand: IntegerGenerator new.
    ^temp

"Brian Foote  6/8/87  Does asOrderedCollection return a copy if it is sent to an
OrderedCollection?"
"Brian Foote  6/8/87  Added the copy above anyway..."

on: aCollection from: firstIndex to: lastIndex
    "Answer an instance of me, streaming over the elements of aCollection
    starting with the element at firstIndex and ending with the one at
    lastIndex."

    ^self on: (aCollection copyFrom: firstIndex to: lastIndex)

    "Brian Foote  6/8/87  Removed the basicNew above (which could not have
worked)..."
```

RandomStream class methods for: examples

The example below returns ten random integers in the range from 1 to 100 inclusive.

RandomStream class methods for: examples

example1

```
"RandomStream example1"

| temp |
temp ← RandomStream on: (Interval from: 1 to: 100).
^(1 to: 10)
    collect: [:n | temp next]
```

SampledStream

SampledStreams are ReadStream-like objects that return the elements of their base collections in a random order. Sampling is performed without replacement. Hence, unlike RandomStreams, **atEnd** will become true after each of the elements of the original base collection has been exhausted.

class name SampledStream
superclass RandomStream
instance variable names
collection
rand
class variable names
pool dictionaries
category Random-Support
hierarchy

```
Object
    Stream
        RandomStream
            SampledStream
```

comment
SampledStreams stream randomly over a copy of the collection given them without replacement...'

SampledStream methods for: accessing

We are **atEnd** when a complete permutation of the original set has been returned. The **next** method uses the method inherited from RandomStream. Note that the **contents** method we inherit from RandomStream is implemented in such as way as to return a permutation of the elements remaining when it is executed by a SampledStream. In particular, if **contents** is executed without calling **next**, a permutation of the entire base class will be delivered.

SampledStream methods for: accessing

```
atEnd
    "Anything left?"

    ^self size ~~ 0

next
    "Return the next element, but remove it from our collection first."

    ^collection remove: super next
```

SampledStream class

class SampledStream class
superclass RandomStream class

SampledStream class methods for: examples

The first example below demonstrates how the contents of a SampledStream generate a permutation of the stream's base class. The second example shows how a SampledStream can be used to randomly select six numbers for the Illinois State Lottery. First, a SampledStream is defined over the numbers from 1 to 44. Next, 6 draws from this stream are collected, converted to a SortedCollection (so that they will print in order) and finally are converted to an Array. Smalltalk can be remarkably concise sometimes...

SampledStream class methods for: examples

```
example1
    "SampledStream example1"

    ^(SampledStream on: #(dog cat mouse gerbil)) contents

quickPick
    "SampledStream quickPick"

    | lotto |
    lotto ← SampledStream on: (1 to: 44).
    ^((1 to: 6)
        collect: [:i | lotto next]) asSortedCollection asArray
```

Plumbing-Support

The classes in this category allow stream-like objects to be strung together into chains that be used to process data. Both read and write steams can be tied together using the fixtures defined in this category. Most of the fixtures given here are write steam style objects. Read stream objects are incorporated into plumbing fixtures (in keeping with the plumbing metaphor used here) using Pump objects to draw values from the read streams and push them down a chain of one or more write streams. Plumbing fixtures are constructed using an easily cascaded connection protocol. The classes in this category are:

Filter	Superclass for other plumbing fixtures
Pump	Pumps drawn values down its sink
Tee	Passes values down any number of sinks
ValueFilter	Transforms objects using a given block
Valve	Passes objects if a block returns true
ValueSupply	Generates values using a given block

The Plumbing-Support objects use a positive pressure, WriteStream-like protocol to pass objects from one fixture object to another. A ReadStream-based approach would have lead to awkwardness at Tee-like objects. With Pump objects to convert between ReadStream and WriteStream-style objects, there was no need to have each fixture object implement both read and write protocols.

This set of objects, perhaps more so than any other set of objects produced as part of the battery simulation, has a high potential for extension. For instance, buffering capabilities could be added to filters, and they could be used as the basis for subclasses that could pass objects from one process to another. Certainly, a set of graphical tools for constructing fixture pipelines is another obvious potential enhancement. The combination of such tools with objects like ValueSupplies, Valves, and Pumps, could constitute the basis for a modest graphical programming language.

Filter

Filter objects are WriteStream-like objects that (usually) pass each value that they receive to another WriteStream-like object that has been designated as their sink. Like Unix filter applications, they will often transform the values they receive before passing them along. This class, Filter, defines the default behavior for most of the plumbing support classes. Straight Filter objects pass the values they receive without making any changes whatsoever to them. More concrete subclasses might change this behavior to do more interesting things.

class name Filter
superclass Stream
instance variable names
    sink
class variable names
pool dictionaries
category Plumbing-Support
hierarchy

```
Object
    Stream
        Filter
            Pump
            Tee
            ValueFilter
            Valve
    ValueSupply
```

comment
Filters allow Stream-like objects and other Filters to be strung together.
They allow nextPut:-based plumbing fixtures to be constructed. Subclasses may add additional functionality so that they do more with the data they receive than merely pass it down the line.

Filter methods for: accessing

The **>>** message designates a given stream or stream like object as the sink, or effluent object. Hence it connects the output of its left-hand operand to the right-hand operand. Since the argument is returned as the result of this message, chains of plumbing fixtures can be strung together using a series of these operators.

The implementation of the **nextPut:** method is such that the object given us will simply be consumed if no sink has been designated. Hence, filters subclasses with useful side effects need not designate special bit-bucket objects to catch their output.

The result of **nextPut:** is, by default, the result of sending **nextPut:** with the argument to the sink stream.

Filter methods for: accessing

>> aSink

"Connect us to the indicated output sink.
Note that we return the given sink so as to
allow connect messages to be strung together."

```
^self sink: aSink.
```

next

"No reading allowed. Filters are WriteStream-like objects."

```
self shouldNotImplement
```

nextPut: anObject

"Write the given object to our output sink. If no sink has been
designated, eat the object."

```
sink isNil ifTrue: [^anObject].
^sink nextPut: anObject
```

sink

"Return our output sink."

```
^sink
```

sink: aSink

"Set our output sink, and return this object as our result."

```
^sink ← aSink
```

Filter class

class	Filter class
superclass	Stream class

Filter class methods for: instance creation

The implementation of **new** we inherited from Steam is overridden here. The inherited method strove to force us to use the **on:** or **with:** methods to create new instances. We restore **new** here.

Filter class methods for: instance creation

new

"It’s okay to use new to create these..."

```
^self basicNew
```

Pump

One of the primary considerations in designing the plumbing support classes was whether to model them after Smalltalk ReadStreams, WriteStreams, or ReadWriteStreams.

An accessing protocol based on ReadStreams would have required the last object in a chain of objects to draw objects from those ahead of it in the chain be sending a **next** message. In the parlance of the plumber, this might correspond to suction or negative pressure.

A WriteStream-based approach, however, requires positive pressure. Objects have to be forced down a chain of fixture objects using **nextPut:** messages. A ReadWriteStream-based approach might have allowed either scheme to be used, or might have required each plumbing fixture object to in effect be a Pump, drawing objects with **next** messages and passing them down the chain using **nextPut** messages.

Pump objects draw objects from a ReadStream-like source and force them down their WriteStream-like sink sides. Pump objects are designed to allow ReadStream-style sources to be connected to WriteStream-style sinks. The objects described here use the WriteStream-based positive pressure metaphor.

class name Pump
superclass Filter
instance variable names
sink

source
category Plumbing-Support
hierarchy

```
Object
    Stream
        Filter
            Pump
            Tee
            ValueFilter
            ValueSink
            Valve
    ValueSupply
```

comment
Pumps draw data from ReadStream-like objects, and send them

down WriteStream-like objects (such as Filters).

Pump methods for: accessing

The **<<** message is used to connect a source object to a Pump. Since this source object is returned as the value of the message, a chain of source objects (but not sink objects) can easily be expressed using this operator.

A pump is activated using the **nextPut** message. This message selector is an amalgam of **next** and **nextPut:**, and results in a **next** being sent to the Pump's source object, with the result therefrom being **nextPut:**'ed down the Pump's output side. (Note the lack of **atEnd** protection in **nextPut**.)

The **nextPutAll** message copies all the objects from the source side of a Pump to the Pump's effluent side. The source is emptied and flushed down the destination pipeline.

The **source** and **source:** methods allow the source fixture to be explicitly accessed. Recall that as a Filter, a Pump inherits all the accessing behavior of Filter objects as well. (The code relies on the user not to make inappropriate use of some of these methods. Pumps might be made more robust by overriding some of this behavior.)

Pump methods for: accessing

```
<< aSource
    "Set our current source object.
    Note that we return the given source so as to
    allow connect messages to be strung together..."

    ^self source: aSource.

nextPut
    "Send the next item down the pike, and return whatever it was as our
    result.  We send a next to our source, and nextPut: the result down our
    sink. "

    ^sink nextPut: source next

nextPutAll
    "Send all we can the pike.  While our source is not atEnd, draw objects
    from it and shove them down our output pipeline."
```

```
[self atEnd]
    whileFalse: [self nextPut]
```

source

"Return our current source object."

```
^source
```

source: aSource

"Set our current source object, and return it as our result."

```
^source ← aSource
```

Pump methods for: testing

If there no source, the pump is always at end. If there is a source object, ask it whether it has any data left.

Pump methods for: testing

atEnd

"Any source data left?"

```
source isNil ifTrue: [^true].
^source atEnd
```

Pump class

class	Pump class
superclass	Filter class

Pump class methods for: examples

The example code below shows how a plumbing fixture might be constructed and used. First, a buffer is created, and a WriteStream is defined over it. Then, a pump is created, and connected to a ReadStream over the Interval from 1 to 10 using the connect source operator **<<**. The output of this Pump is then connected to a ValueFilter that doubles each value that passes through it. This is connected to a Valve that only passes objects that are evenly divisible by 4. The objects that are passed by this Valve flow into the result stream constructed above. Once this fixture is constructed, all the values from its input side are pumped to its output side using the **nextPutAll** message. The contents of the result stream are then returned. This result will be the numbers between 2 and 20 divisible by 4.

Pump class methods for: examples

```
example1
	"Pump example1"

	| result fixture |

"We'll dump our results on this array, to which we will fit a stream..."
	result ← WriteStream on: (Array new: 100).

"Build a fixture.  Start with a pump.  Hook a stream on an interval
into it as a data source.  Run its output into a ValueFilter, which
will double each value, and a divisible by four Valve, then into
the result stream..."
	fixture ← Pump new.
	fixture << (ReadStream on: (Interval from: 1 to: 10)).
	fixture
		>> (ValueFilter using: [:value | value * 2])
		>> (Valve using: [:value | value \\ 4 = 0])
		>> result.

"The fixture is all constructed, so now let's use it.
Run all the data from our input stream through
the pipeline constructed above into the result stream.
Return the contents of the result stream..."
	fixture nextPutAll.
	^result contents
```

Tee

Tee objects are like normal Filters, except that instead of using a single sink, they pass each object they receive to multiple sinks.

class name Tee
superclass Filter
instance variable names
sink
class variable names
pool dictionaries
category Plumbing-Support
hierarchy

```
Object
    Stream
        Filter
            Pump
            Tee
            ValueFilter
            ValueSink
            Valve
    ValueSupply
```

comment
Tees are output Filters that allow multiple output sinks.
(If this facility proves really useful, it can be added to Filter.)

Tee methods for: accessing

To add multiple outputs to sink, only two messages needed to be overridden. The *sink* instance variable was made to refer to a Collection, and **>>** and **nextPut:** were changed to allow multiple connections and multiple writes.

Tee methods for: accessing

>> aSink

```
"Add this sink to our collection..."

self sink isNil ifTrue: [self sink: (OrderedCollection new: 4)].
self sink addLast: aSink.
^aSink
```

nextPut: anObject

```
"Do writes to all our effluent ports..."

self sink do: [:aSink | aSink nextPut: anObject].
^anObject
```

ValueFilter

Class ValueFilter overrides Filter to allow a user supplied block to modify each object passed to a Filter before it is passed to subsequent plumbing fixture stages. The combination of Filters and Smalltalk blocks is a very powerful one. The ability to pass a ValueFilter an arbitrary block removes the need for special purpose subclasses to make specialized value transformations.

The role performed by this class in analogous to that performed by the **collect:** messages in the Collection hierarchy.

class name ValueFilter
superclass Filter
instance variable names
sink

block
class variable names
pool dictionaries
category Plumbing-Support
hierarchy

```
Object
    Stream
        Filter
            Pump
            Tee
            ValueFilter
            ValueSink
            Valve
    ValueSupply
```

comment
ValueFilters transform the input they receive using a designated block, and send the result to their output sinks...

ValueFilter methods for: accessing

ValueFilters allow the block to be reset using a **block:** message. The **nextPut:** method checks whether a block is defined, and if one is, passes the object it was passed to the block as its argument. This value is then passed down the pipeline using the standard Filter method for doing this. If no block is defined, ValueFilters revert to behaving like a straight Filters.

ValueFilter methods for: accessing

```
block: aBlock
    "Set (or reset) our transformation..."

    ^block ← aBlock

nextPut: anObject
    "Return the value of running the given object through our block..."

    block isNil
        ifTrue: [^super nextPut: anObject]
        ifFalse: [^super nextPut: (block value: anObject)]
```

ValueFilter class

class	ValueFilter class
superclass	Filter class

ValueFilter class methods for: instance creation

The **using:** instance creation method allows a ValueFilter's block to be specified as a new ValueFilter is created. This block must be a single argument block. Each time an object is passed through a ValueFilter, it will be passed through this block before being sent down the rest of the pipeline.

ValueFilter class methods for: instance creation

```
using: aBlock
    "Set our value transformation block..."

    | filter |
    filter ← super new.
    filter block: aBlock.
    ^filter
```

Valve

Valve objects pass objects through to their output sides only if they pass a designated test. This test is provided by the user in the form of a boolean single argument block. Each time an object is passed to a Valve, the Valve passes it through the given block. If the result of the block is True, the object is passed to the Valve's output side. If the result is False, nothing is passed to the Valve's output. The combination of Smalltalk blocks and conditional Valves allows arbitrary tests to be built into plumbing fixtures.

The role performed by this class in analogous to that performed by the **select:** messages in the Collection hierarchy.

class name Value
superclass Filter
instance variable names
sink

block
class variable names
pool dictionaries
category Plumbing-Support
hierarchy

```
Object
    Stream
        Filter
            Pump
            Tee
            ValueFilter
            ValueSink
            Valve
    ValueSupply
```

comment
Valve objects pass data through to their output sinks
only if a given block evaluates to true...

Valve methods for: accessing

The **nextPut:** method for Valves passes each value it is passed through the designated test block. If this block returns True, the Value passes the object to the its effluent side. If the test is False, it merely returns.

Valve methods for: accessing

block: aBlock

```
"Set (or reset) our predicate block.  This should be a one argument
block, which will receive the current output object..."

^block ← aBlock
```

nextPut: anObject

```
"Return the value of running the given object down our sink only if we
have a predicate block, and it is true.  Otherwise return the object itself..."

block isNil ifTrue: [^anObject].
(block value: anObject)
    ifTrue: [^super nextPut: anObject].
^anObject
```

Valve class

class	Valve class
superclass	Filter class

Valve class methods for: instance creation

The **using:** method allows a boolean single value block to be designated as the test block for a Valve as the Valve is created.

Valve class methods for: instance creation

using: aBlock

```
"Set our test block..."

| filter |
filter ← super new.
filter block: aBlock.
^filter
```

ValueSupply

ValueSupply objects are the only ReadStream-style objects in the Plumbing-Supply system category. Like ValueFilters and Valves, they exploit the power of Smalltalk blocks to allow an arbitrary block to be invoked whenever a value is needed. This block must be a zero argument block.

```
class name      ValueSupply
superclass      Object
instance variable names
                block
class variable names
pool dictionaries
category        Plumbing-Support
hierarchy
                Object
                    Stream
                        Filter
                            Pump
                            Tee
                            ValueFilter
                            ValueSink
                            Valve
                    ValueSupply
comment
This ReadStream-like object evaluates a given block in response
to next requests...
```

ValueSupply methods for: accessing

The **next** method queries the ValueSupply's block for a value and returns it.

ValueSupply methods for: accessing

```
    block: aBlock
        "Set (or reset) our value generating block..."

        ^block ← aBlock

    next
        "Return the result of evaluating our block..."

        ^block value
```

ValueSupply class

class ValueSupply class
superclass Object class

ValueSupply class methods for: instance creation

The **using:** instance creation method allows a zero argument block to be specified for a ValueSupply as it is created.

ValueSupply class methods for: instance creation

```
using: aBlock
    "Set our value generating block..."

    | filter |
    filter ← super new.
    filter block: aBlock.
    ^filter
```

Accessable-Objects

The classes in the Accessable-Object system category allow dictionary-like objects to be accessed using the conventional record-style accessing protocol, and allow instance variables to be accessed using the dictionary-style accessing protocol. The effect is to allow the attribute list type programming style seen in some Lisp dialects and the record style accessing schemes seen in Pascal and Simula style languages to be used when appropriate on individual objects. The objects in this category are:

AccessableObject	Allow dictionary access to a "record"
AccessableDictionary	Allow "record" access to a dictionary

AccessableObject

Accessable objects are objects that allow dictionary-style access to all their instance variables, along with record-style access to a built in dictionary. Hence, instance variables can be accessed using **at:** and **at:put:**, as well as the standard record-style access protocol (**name** and **name:**).

Both access styles are provided without any need to explicitly define additional accessing methods. The record-style access method is rather slow however, and should be overridden when efficiency is an important consideration.

If **name:** or **at:put:** storage attempt is made and no instance variable with the given name exists, an entry is made for the given selector in the AccessableObject's item dictionary. Thereafter, this soft instance variable may be accessed using <u>either</u> <u>access</u> <u>method</u>. In this way, uniform access to hard and soft fields is provided, and Smalltalk's Simula/Lisp dualism is bridged. AccessableObjects provide a way of adding associations to objects in a manner similar to that provided by Lisp's property list mechanisms.

class name AccessableObject
superclass Object
instance variable names
items
class variable names
pool dictionaries
category Accessable-Objects
hierarchy

```
Object
    AccessableObject
        BatteryItem
            SternbergTask
            ToneOddball
            WordOddball
        BatteryParameters
            SternbergTaskParameters
            ToneOddballParameters
            WordOddballParameters
        ListHolder
        ProtocolItem
```

AccessableObject methods for: instance initialization

All AccessableObjects respond to **initialize** and at least do nothing.

AccessableObject methods for: instance initialization

```
    initialize
        "As a default, do nothing..."

        ^self
```

AccessableObject methods for: accessing

AccessableObjects allow Dictionary style access to AccessableObject instance variables and soft (item dictionary) fields through the **valueAt:** family of messages. The **at:** family of messages employs these to do its work.

The **items** and **items:** accessing messages allow explicit, efficient access to an AccessableObject's item dictionary.

The **size** of an AccessableObject is defined to be the size of the items dictionary plus the number of instance variables the object has.

AccessableObjects use the **value** prefix before the **at** family messages so that the **at** messages might be overridden, when necessary, without removing the accessing capabilities. (These are "basic" accessing methods, in a sense.)

The **valueAt:ifAbsent:** method is the main accessing method. It first sees if we have an instance variable with the name given it. If so, we return this value. If there is no variable by this name, we look for the given key in the item dictionary. If this search fails, we execute an error block. For **valueAt:**, this is the standard Dictionary failure message **errorKeyNotFound**.

The **valueAt:put:** method works in a similar fashion. It first allocates an item dictionary, if need be. It then sees if an instance variable with the given name exists. If so, it calls upon **variableAt:put:** to store the value. If there is no instance variable with the given name, an entry is made in the items dictionary.

AccessableObject methods for: accessing

```
at: key
    "Return the object associated with the given key..."

    ^self valueAt: key

at: key ifAbsent: aBlock
    "Allow graceful recovery if an item is not defined..."

    ^self valueAt: key ifAbsent: aBlock

at: key put: value
    "Store the indicated value at the designated place in our item
    dictionary... "

    ^self valueAt: key put: value

items
    "Return our item list..."

    ^items

items: itemList
    "Set our item list..."

    ^items ← itemList

size
    "Let's say our size is the size of our item dictionary plus our number of
    instance variables..."

    ^self items size + self instVarNames size

valueAt: key
    "Return the object associated with the given key..."

    ^self valueAt: key ifAbsent: [self errorKeyNotFound]

valueAt: key ifAbsent: aBlock
    "Allow graceful recovery if an item is not defined..."

    ^self variableAt: key ifAbsent: [^self items at: key ifAbsent: aBlock]

valueAt: key put: value
    "Store the indicated value at the designated place in our item
    dictionary, unless there is an instance var by that name..."

    items isNil ifTrue: [items ← IdentityDictionary new: 16].
    (self hasVariableNamed: key)
```

```
ifTrue: [^self variableAt: key put: value]
ifFalse: [^items at: key put: value]
```

AccessableObject methods for: instance variable access

The methods in this protocol category provide the instance variable access mechanisms.

The **allInstVarNames** method exploits a class method to return a collection of strings representing all the instance variables defined for a given AccessableObject (including its superclasses).

The **hasVariableNamed:** method is a predicate which is true if we have an instance variable with the given name. (Again, instance variables defined by the superclasses count.)

The **instVarNames** method returns only those instance variables added by a given class itself, not those defined in any superclasses.

The **variableAt:** method allows an instance variable to be accessed using a Symbol or String as a selector. The **variableAt:ifAbsent:** method allows a failure block to be given.

The **variableAt:put:** method allows an arbitrary instance variable to be stored into given a symbolic key.

The **variableIndex:** method converts an instance variable name into an index suitable for the basic **instVarAt:** and **instVarAt:put:** methods defined in class Object.

AccessableObject methods for: instance variable access

allInstVarNames
"Define a shorthand for this class method..."

```
^self class allInstVarNames
```

hasVariableNamed: name
"Say whether we have a variable by the given name..."

```
^(self variableIndex: name) ~= 0
```

```
instVarNames
	"Define a shorthand for this class method..."

	^self class instVarNames

variableAt: name
	"Return the named value..."

	| index |
	index ← self variableIndex: name.
	index = 0 ifTrue: [self error: 'Bad instance variable name...'].
	^self instVarAt: index

variableAt: name ifAbsent: aBlock
	"Return the named value..."

	| index |
	index ← self variableIndex: name.
	index = 0 ifTrue: [^aBlock value].
	^self instVarAt: index

variableAt: name put: value
	"Set the named instance variable to the indicated value..."

	| index |
	index ← self variableIndex: name.
	index = 0 ifTrue: [self error: 'Bad instance variable name...'].
	^self instVarAt: index put: value

variableIndex: name
	"Return the instance variable index for this name, or zero..."

	^self class allInstVarNames indexOf: name asString.
```

AccessableObject methods for: error interception

AccessableObjects use **doesNotUnderstand:** to convert record-style accesses to AccessableObjects to instance variable or item dictionary probes.

To accomplish this, they first pick up the selector from the message we we sent. Then they pick up the name for the selector by copying it without the trailing colon (if present). Next, they ask the message for its argument count. After this, they test to see whether an instance variable with the given name is defined for this object. If one is, AccessableObjects use the number of arguments to determine whether they should do a read or a store on this instance variable.

Note that this method of performing instance variable access is much less efficient than if explicit instance variable access methods are defined for a given object.

If no instance variable with the given name is defined, an AccessableObject asks the item dictionary itself if it responds to the message we were passed. If it does, we pass this message along to the item dictionary. (This allows some of the functionality that would be available were AccessableObjects full fledged Dictionaries to be exploited. A **keysDo:** to an AccessableObject, for example, will be performed by the item dictionary. This will often, but not always, do the expected thing. For example, instance variables will not be included among the keys processed using this scheme. When this is the effect desired, one should convert the AccessableObject into an AccessableDictionary, and then request the desired service.)

If the item dictionary cannot respond to the given selector, an item dictionary access is attempted. If this fails, the **doesNotUnderstand:** method of AccessableObject's superclass (Object) is invoked.

AccessableObject methods for: error interception

```
doesNotUnderstand: aMessage
    "Refer messages we don't understand to our item dictionary..."

    | selector name args |
    selector ← aMessage selector.
    name ← (selector copyWithout: $:) asSymbol.
    args ← aMessage arguments.
    (self hasVariableNamed: name)
        ifTrue: [args size = 1
                ifTrue: [^self variableAt: name]
                ifFalse: [self variableAt: name put: (args at: 1)]].
    (items respondsTo: selector)
        ifTrue: [^items perform: selector withArguments: args].
    args size = 1
        ifTrue: [^self valueAt: name put: (args at: 1)]
        ifFalse: [^self valueAt: name ifAbsent: [^super doesNotUnderstand: aMessage]]
```

AccessableObject methods for: conversion

It is occasionally prudent to convert an AccessableObject into an AccessableDictionary. The resulting AccessableDictionary will share all the underlying objects that were accessed by the AccessableObject, and will allow standard Smalltalk Dictionary manipulations (such as **keysDo:**) to refer to all items referenced by the original AccessableObject, whether they were referred to by instance variables or item dictionary entries. Note in particular that **at:** and **at:put:** style references will incur much less overhead when made using an AccessableDictionary that they would if made to an AccessableObject.

The **asAccessableDictonary** method works as follows. First, an AccessableDictionary with 10 extra slots is created. Then all the items in our item dictionary are copied to it. Then make entries for each of the instance variables are made, and these values are copied as well.

AccessableObject methods for: conversion

```
asAccessableDictionary
    "Return an AccessableDictionary with the same contents as us..."

    | dict |
    dict ← AccessableDictionary new: self size + 10.
    self items~~ nil ifTrue:[self items keysDo: [:key | dict at: key put: (self at: key)]].
    self allInstVarNames do: [:name | dict at: name asSymbol put: (self at: name
asSymbol)].
    ^dict
```

AccessableObject methods for: private

This standard complaint was copied from Dictionary.

AccessableObject methods for: private

```
errorKeyNotFound
    "Make the obvious complaint..."

    self error: 'key not found'
```

AccessableObject class

class AccessableObject class
superclass Object class

AccessableObject class methods for: instance creation

AccessableObject class methods for: instance creation

```
new
    "Slip in a default instance initialization message..."

    | anObject |
    anObject ← super new.
    anObject initialize.
    ^anObject
```

AccessableObject class methods for: examples

The example below shows how an AccessableObject can be created and used as an extensible record object. The #dog and #cat entries are added using the record-style accessing protocol. The record-style access to these soft fields is then tested. Finally, the example iterates over the soft fields using **keysDo:**.

AccessableObject class methods for: examples

```
example
    "AccessableObject example"

    | temp |
    temp ← AccessableObject new.
    temp dog: 'Fido'.
    temp cat: 'Tabby'.
    Transcript print: temp dog; cr.
    Transcript print: temp items; cr.
    temp keysDo: [:key | Transcript print: key; cr].
    Transcript print: (temp variableAt: #items); cr.
    Transcript endEntry
```

AccessableDictionary

AccessableDictionaries are Dictionaries that allow their fields to be accessed using record-style (**name** and **name:**) conventions. They complement AccessableObjects by providing a means of bringing the full power of Smalltalk Dictionary objects to bear on hybrid AccessableObject (which may be constructed using some hard and some soft fields). The addition of this uniform reference capability to Dictionaries is a worthy extension in its own right.

class name AccessableDictionary
superclass Dictionary
instance variable names
tally
class variable names
pool dictionaries
category Accessable-Objects
hierarchy

```
Object
    Collection
        Set
            Dictionary
                AccessableDictionary
                    DataDictionary
                    ParameterDictionary
```

AccessableDictionary methods for: accessing

We provide these methods for compatibility with AccessableObjects.

AccessableDictionary methods for: accessing

```
items
    "Say that we are one big item list..."

    ^self

items: itemList
    "Say that we are one big item list..."

    ^self error: 'AccessableDictionaries should not reset their item lists...'

valueAt: key
    "Just send this to ourself..."

    ^self at: key
```

valueAt: key ifAbsent: aBlock

```
"Just send this to ourself..."

^self at: key ifAbsent: aBlock
```

valueAt: key put: value

```
"Just send this to ourself..."

^self at: key put: value
```

AccessableDictionary methods for: error interception

This is a simplified variation of the method found in AccessableObject. It converts record-style accesses into dictionary accesses. First, the selector and argument count for the message sent to the AccessableDictionary are determined. The method then determines whether the selector is a keyword selector. If so, the message is assumed to be a request to store a value. The selector is copied without colons, and a dictionary entry entry is created. If this is a unary selector, a probe is attempted. If this probe fails, an error message is given.

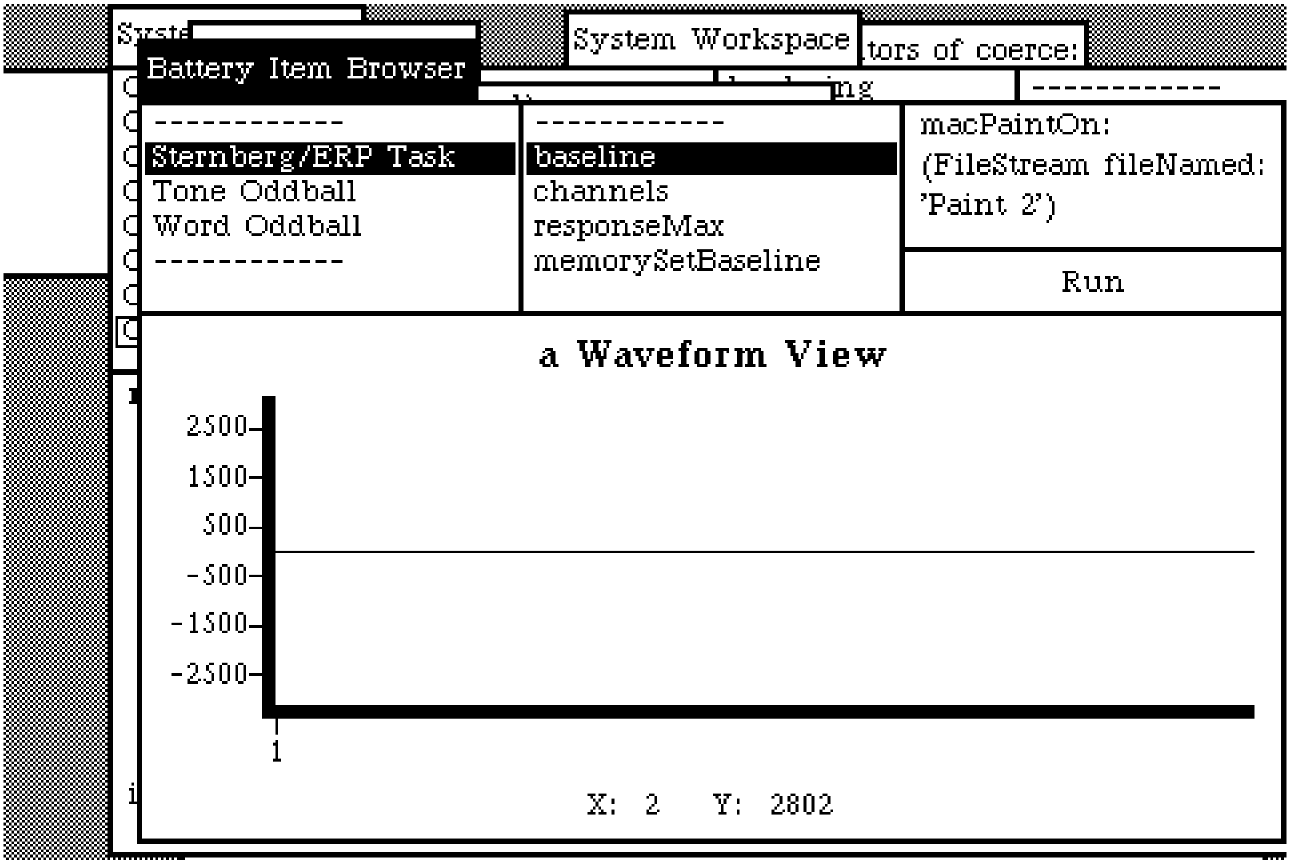
AccessableDictionary methods for: error interception

doesNotUnderstand: aMessage

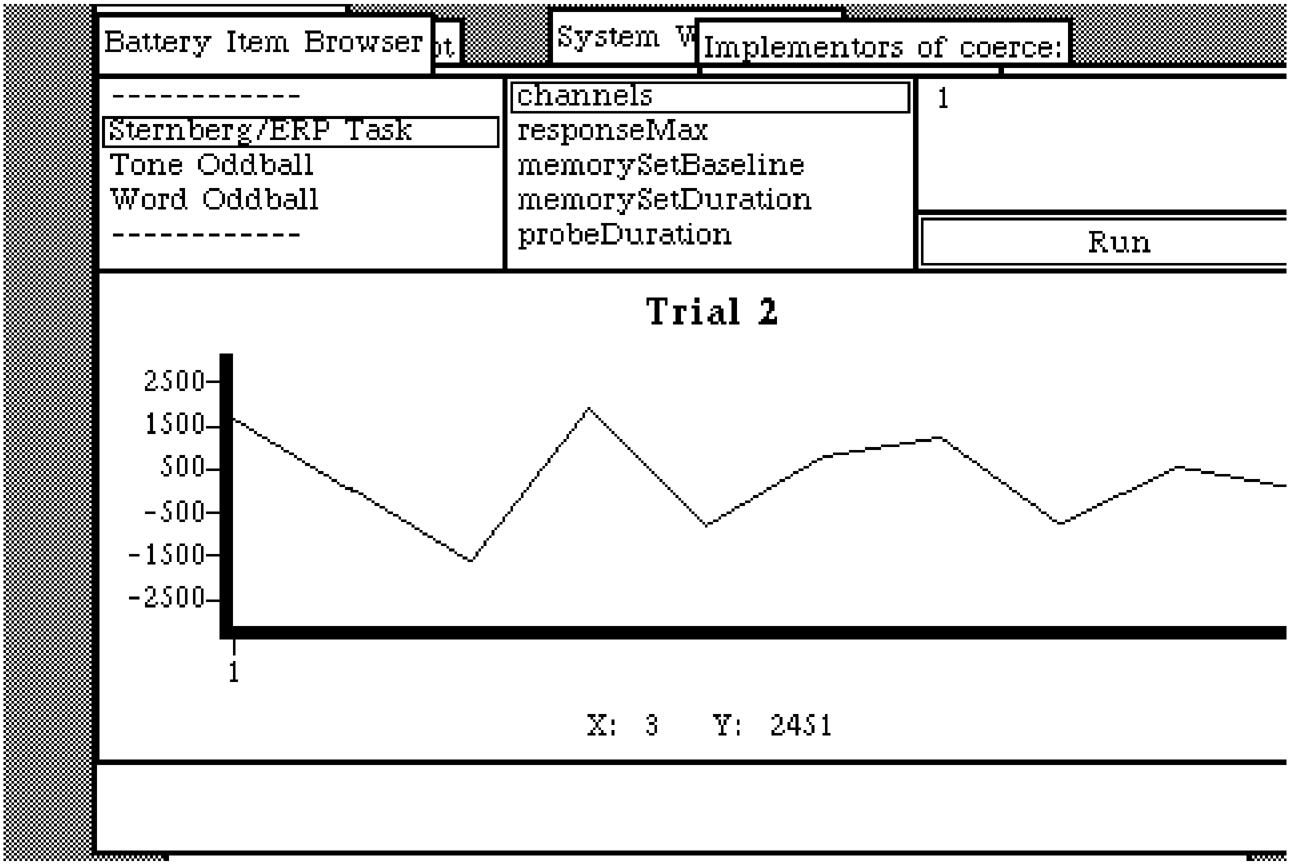
```
"Convert accessing messages to dictionary accesses..."

| selector args |
selector ← aMessage selector.
args ← aMessage arguments.
selector isKeyword
    ifTrue:
        [selector ← (selector copyWithout: $:) asSymbol.
        ^self at: selector put: (args at: 1)]
    ifFalse: [^self at: selector ifAbsent: [^super doesNotUnderstand: aMessage]]
```

Chapter V -- A Tour of the Battery Simulation

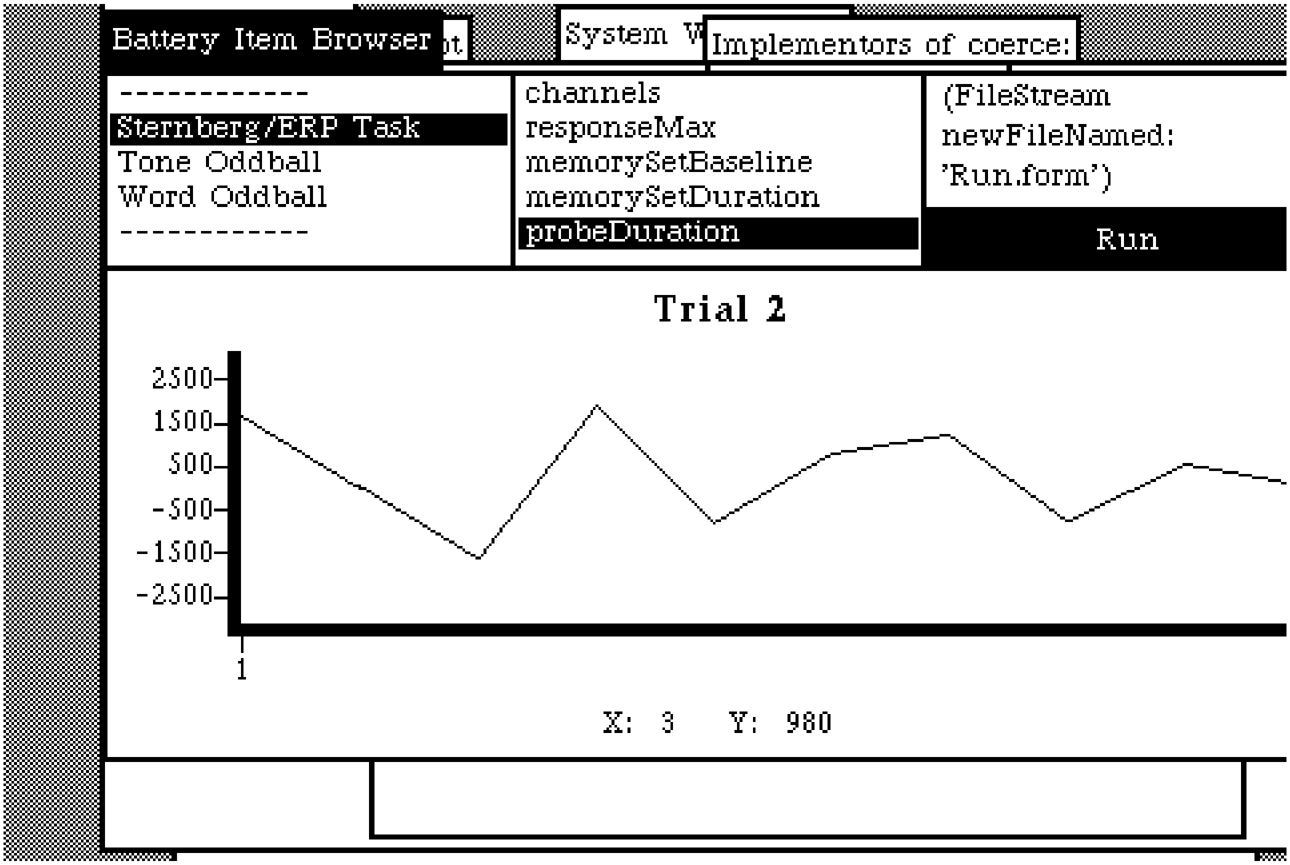
This chapter shows how the user interacts with the simulated battery using the Battery Item Browser, and how the simulation appears when a battery item is running.



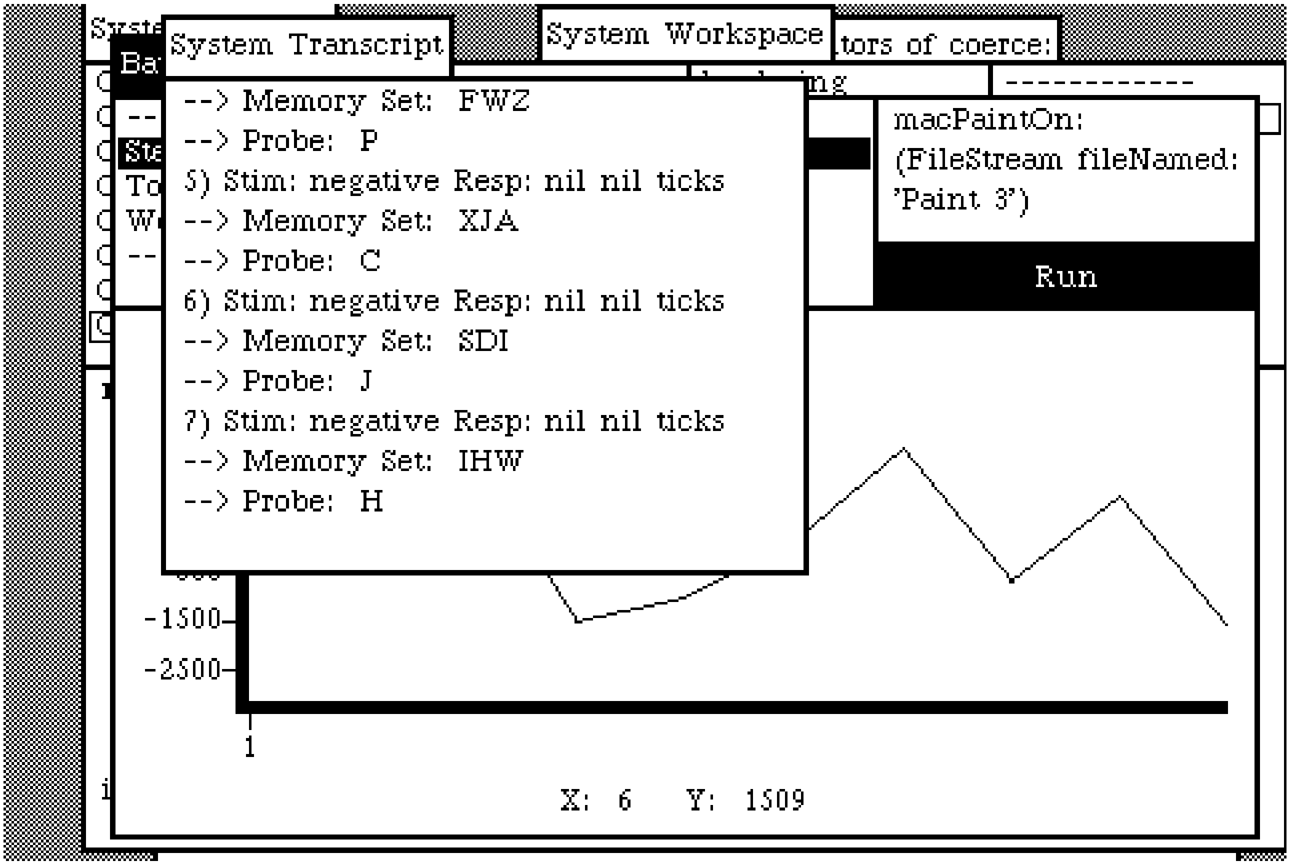
The figure above shows he Battery Item Browser before any battery items have been run. The the pane at the upper left of the browser gives a list of the available battery items. When an item is selected, a list of its parameters appears in the center pane.



When a parameter value is selected, its value is displayed in the pane at the upper right of the battery browser. The parameter value pane is similar to a standard Smalltalk-80 code view.



This example shows how the Battery Browser appears while a block of trials is running. Once the experimenter is satisfied with his or her parameter settings, a block of trials may be initiated by pressing the "Run" switch. While a block of trials is running, the "Run" switch is highlighted, and a waveform average display is continually shown on the screen. The amplitude of the waveform may be interactively examined using the mouse. The a crosshair cursor is displayed when the mouse is within the waveform frame to indicate this.



The battery simulation uses the Smalltalk system transcript to simulate the original battery's more elaborate VT100 runtime informational display. When the transcript is updated, it pops in front of the Battery Browser. On a system with a larger screen area, the transcript could be kept off to the side.

Chapter VI -- Discussion

This chapter discusses a number of issues raised by this investigation. The bulk of the discussion is centered on the emergence and use of *object-oriented frameworks*, and their impact on the design process and the software lifecycle. Later sections discuss the impact of object-oriented techniques in general, and the Smalltalk-80 system in particular, on the design and implementation of realtime laboratory applications.

The chapter begins with a discussion of object-oriented frameworks and environments. This section explains what distinguishes them from conventional programming techniques. Next, the problems associated with skeleton programs are discussed. This is followed by a lengthy examination of why software design, which is difficult under any circumstances, is particularly difficult in the presence of changing requirements. Subsequent sections discuss designing to facilitate change, and the tension between specificity and generality.

These discussions are followed by an examination of certain software lifecycle issues. The so-called maintenance phase and the question of whether to reuse or reinvent software are covered in these sections.

The final parts of this chapter examine object-oriented programming and realtime applications, as well as the problems of designing general data structures in Smalltalk.

Object-Oriented Frameworks

An object-oriented framework is a set of classes that provide the foundation for solutions to problems in a particular domain. Individual solutions are created by extending existing classes and combining these extensions with other existing classes.

The Battery simulation described herein is a framework for constructing realtime psychophysiological application programs. MacApp [Apple 85] is a framework for constructing Macintosh application programs. It is in effect a generic application program that provides standard Macintosh user interface and document handling capabilities. The Lisa Toolkit [Apple 83] was an earlier package that used object-oriented techniques to integrate applications into the Lisa desktop environment. The Smalltalk-80 Model-View-Controller triad (MVC) is a framework for constructing Smalltalk-80 user interfaces [Goldberg 84].

Frameworks are more than well written class libraries. A good example of a set of utility class definitions is the Smalltalk Collection hierarchy. These classes provide ways of manipulating collections of objects such as Arrays, Dictionaries, Sets, Bags, and the like. In a sense, these tools correspond to the sorts of tools one might find in the support library for a conventional programming system. Each component in such a library can serve as a discrete, stand-alone, context independent part of a solution to a large range of different problems. Such components are largely application independent.

A framework, on the other hand, is an *abstract design* that much more closely resembles a skeleton program depicting a solution to a specific kind of problem. It is a template for a family of solutions to a set of related problems. Where a class library contains a set of components that may be reused individually in arbitrary contexts, a framework contains a set of components that must be reused together to solve a specific instance of a certain kind of problem.

One should not infer that frameworks are useful only for reusing mainline application code. The abstract designs of library components as well as those of skeletal application code may serve as the foundations for frameworks. The ability of frameworks to allow the extension of existing library components is in fact one of their principal strengths.

It is important to distinguish between using library components in the traditional fashion and using them as part of the basis for a framework. For example, most of the classes in Chapter V (the battery library) of this document are used by the battery simulation primarily as traditional, discrete library components. However, the classes in Chapter IV (the battery framework) illustrate a set of three concrete applications derived from a generic psychophysiological experimental design. Each application was created by defining application specific subclasses that selectively overrode and extended the behavior of the abstract battery design. A finished application is comprised of a cooperating set of specialized framework objects supplemented by library and system components. The view and controller classes in the Interface-Battery category are examples of system components that have been specialized by the battery simulation.

Recall that one way to look at a framework is as an abstract design. Such a design is extended and made concrete via the definition of new subclasses. Each method that a subclass adds to such a framework must abide by the internal conventions of its superclasses.

Another type of framework is a collection of abstract component designs. A library of components will, in this case, supply at least one, and perhaps several components that fit each abstract design.

The major difference between using object-oriented frameworks and using component libraries is that the user of a component need understand only its external interface, while the user of a framework must understand the internal structure of the classes being extended by inheritance. Thus, components are "black boxes" while frameworks are "white boxes". Clearly, inheritance-based frameworks require more training to use and are easier to abuse than component-based frameworks, but they allow application-dependent algorithms to be recycled more easily.

A few examples from the battery simulation should serve to illustrate these distinctions. The abstract classes BatteryItem and BatteryParameters constitute the roots of an inheritance framework that defines generic experiments. The subclasses WordOddball, ToneOddball, and SternbergTask, along with WordOddballParameters, ToneOddballParameters, and SternbergTaskParameters define three concrete applications derived from this framework.

The abstract classes Stimulus and StimulusGenerator define a generic stimulus handling mechanism. The subclasses WordGenerator, ToneGenerator, and SternbergDisplayGenerator, along with WordStimulus, ToneStimulus, and SternbergStimulus define concrete realizations of these designs for each battery application that has been defined. Hence, the two hierarchies built around Stimulus and StimulusGenerator are inheritance frameworks as well. The four hierarchies rooted in BatteryItem, BatteryItemParameters, Stimulus, and StimulusGenerator can hence be thought of as together comprising a framework for constructing experiments.

However, the relationship between the BatteryItem and StimulusGenerator hierarchies merits additional scrutiny. Subclasses of BatteryItem do not have direct access to the states of their stimulus generation objects. They instead own instances of an appropriate kind of StimulusGenerator. They thus have access to these objects only via their external protocols. Hence, even though the class hierarchies for stimulus handling parallel those used to define the battery items, this need not be the case in general. In fact, <u>any</u> object that met the protocol assumptions made in a given battery item for its stimulus generator would, as a result of polymorphism, work correctly with that battery item. The battery items in this case define a component framework, in which any object that meets a given abstract design may replace any other.

If the relationship between parts of a framework can itself be defined in terms of an abstract protocol (or signature), instead of using inheritance, then the generality of all the parts of the framework is enhanced. In fact, as the design of a system becomes better understood (as it evolves) more component-based relationships should emerge to replace inheritance-based ones. One can think of such component-based relationships as an ideal towards which system components should be encouraged to evolve.

The stimulus hierarchies discussed above are examples of just this sort of evolution. In the CPL battery, as well as in earlier versions of the battery simulation, the stimulus generation code was intertwined tightly with other battery item control code. It became clear as I worked with this code that separating the stimulus generation code from the rest of the battery items would increase the modularity and clarity of both the resulting hierarchies. In addition, even though there is still a one-to-one correspondence between the concrete subclasses in each hierarchy, the relationship among these objects need not stay this way. Since the stimulus generation objects are now components of battery items, future items may mix and match these as they please.

Object-oriented inheritance allows extensions to be made while leaving the original code intact. The original root objects of a framework may also serve as the basis for any number of concrete extensions. A framework may be structured as a hierarchy of increasingly more specific subclasses. In contrast, a purely component-based reuse strategy requires that application code that orchestrates application independent components either be conditionalized or rewritten for each new application.

A framework may, in ideal cases, evolve from the sort of white box structure described above into a gray or even black box structure. It may do so in those instances where overridden methods can be replaced by message sends to black box parameters. Examples of such frameworks are the sorting routines that take procedural parameters seen in such conventional systems. Where it is possible to factor problems in this fashion, it is certainly desirable to do so. Reducing the coupling necessary between framework components so that the framework itself works with any plug-compatible object increases its cohesion and generality. A number of authors have written on the desirability of using component-decomposition as an alternative to inheritance where possible (see [Halbert 87] and [Johnson 88] for instance). It should be a goal of the framework architect to allow a given framework to evolve into a component framework. The route by which a given inheritance framework will evolve into a component framework will not always be obvious, and many (most?) frameworks will not complete the journey from skeleton to component frameworks during their lifetimes.

White box inheritance frameworks should be seen as a natural stage in the evolution of a system. They may be intermediate stages in the evolution of the system, or permanent fixtures in it. The important thing to realize about them is that by providing a middle ground between task specific and general realizations of a design, white box inheritance frameworks provide an indispensable path along which applications may evolve.

Barbara Liskov, in a keynote address given at the OOPSLA '87 conference in Orlando, distinguished between inheritance as an implementation aid (which she dismissed as unimportant) and inheritance for extending the abstract functionality of an object. Liskov claims that in the later case, only the abstract specification, and not the internal representation of the parent object should be inherited. She was in effect advocating that only the black box framework style described above should be employed. Such a perspective ignores the value of white box frameworks, particularly in the face of changing requirements.

A failure to confront the challenges of design evolution is characteristic of most of the work that has been done on programming methodologies. It is not my intention to fault researchers in this area for this, however. The problems of system design are difficult enough without adding the enormous additional complication of moving target requirements.

One way of characterizing the difference between inheritance and component frameworks is to observe that in inheritance frameworks, the state of each instance is implicitly available to all the methods in the framework, much as the global variables in a Pascal program are. In a component framework, any information passed to constituents of the framework must be passed explicitly. Hence, an inheritance framework relies, in effect, on the intra-object scope rules to allow it to evolve without forcing it to subscribe to an explicit, rigid protocol that might constrain the design process prematurely.

It is possible, albeit difficult, to design good class libraries and frameworks in a top-down fashion. More frequently, good class libraries and frameworks <u>emerge</u> from attempts to solve individual problems as the need to solve related problems arises. It is through such experience that the designer is able to discern the common factors present in solutions to specific problems and construct class hierarchies that reflect these commonalities. It is the ability of inheritance hierarchies to capture these relationships as they emerge that makes them such powerful tools in environments that must confront volatile requirements.

Much of the remainder of this section looks at the impact of frameworks on the software design process and the software lifecycle, with a particular emphasis on the problems surrounding application environments characterized by rapidly changing requirements. We will compare the framework-based approach with conventional solutions to such problems, such as application skeletons and subroutine libraries, and examine those areas addressed by frameworks where no clearly recognized alternative methodologies currently exist.

Environments

An object-oriented application construction environment, or environment, is a collection of high level tools that allow a user to interact with an application framework to configure and construct new applications. Environments are referred to as Toolkits in [Johnson 88]. Examples of environments are Alexander's Glazier system for constructing Smalltalk-80 MVC applications [Alexander 87], and Smith's Alternate Reality Kit [Smith 87]. A framework for a given application domain can often serve as the basis for the construction of tools and environments for constructing and managing applications.

Getting Skeletons Back in the Closet

One has only to examine the state of practice in many research application domains to realize how much they need better tools and techniques. There are certainly some problems with the approaches taken in the research described herein. To put things in perspective, it might serve us well to look at how much worse the status quo is.

Let's examine the ways in which realtime psychophysiological application programs developed in a typical laboratory environment. (The author has spent a considerable amount of time working in this domain. See [Heffley 84].)

At the University of Illinois' Cognitive Psychophysiology Laboratory, research applications are normally written by scientists and graduate students, using Fortran 66 and a structured preprocessor. Realtime device access, and time critical functions are performed using a large library of fast assembly language subroutines. (See [Foote 85].) Some might ask whether a language such as Pascal or C might be a better choice for this sort of application programming. In fact, Fortran retains a number of advantages over these other languages for scientific programming. These include de facto support for separate compilation, good support for subroutines that manipulate arbitrarily dimensioned multidimensional arrays, and very good floating point support. In any case, so far so good.

A given group of students working in such a research environment will exhibit tremendous diversity in their levels of programming ability. Some students will have extremely strong computer backgrounds, others will have minimal programming backgrounds. One result of this is that novel application programs are developed by those with the better computer skills, and become templates, or skeleton programs, upon which others build.

The person writing such a skeleton program is seldom interested in solving anything more than the problem at hand. The program is only a tool for conducting the current experiment. A programmer in a research environment will often be in a poor position to plan for future version of a program, since research is often a somewhat fuzzy, exploratory enterprise. The initial version of a given program might be a short, spare shell that performs the rudiments of a given experimental design and nothing more. Since these initial programs are usually thought of (at least at the time) as one-shot, disposable programs, they are frequently underdocumented or undocumented. (The question of what constitute a truly well documented program is a very interesting one I will not try to address here.) Still, things have not yet gotten unmanageable. The production of disposable programs may be a reasonable practice when applications will consistently be short and trivial, and where successive applications will not be required to try to build on previous ones.

Trouble can begin when new problems that resemble the one for which an original program was written in some respects (but not others) arise. This is a familiar situation in most application domains, but occurs with much greater frequency in some research programming domains, again, because such is the nature of research. When this point is reached, the programmer is faced with the following choices:

He or she can write a new program "from scratch", perhaps based in some respects on the original program. This alternative is merely the generation of a new "disposable" program.

The programmer can generalize the existing program to accommodate both the old and new requirements. In some instances, the existing program can be made to accommodate the new requirements merely by altering overly specific code. (An example of this is a program written to collect, say, 2 channels of data, when it could nearly as easily have been written to collect "n" channels.) I call this approach "parameterization".

More frequently, accommodating multiple requirements in a given program will require "conditionalization", the addition of flag parameters or other variant tag tests that direct the flow of control to code that implements either the old or new requirements.

Global variants can be eliminated very elegantly using an object-oriented inheritance hierarchy. The base class containing tag checks becomes an abstract superclass. At each point where a tag is tested, a method dispatch to self is performed instead. (Such variant elimination might even be automated.) Each variant becomes a subclass of the new abstract class. The methods in these subclasses encompass only the code that distinguishes them from the abstract superclass. The result is much more readable, since the code that distinguishes each variant from the common ancestor is collected together, rather than scattered among case statements all over the parent module. Variant elimination and the generation of such abstract superclasses thus promote the emergence of frameworks.

The final alternative is to <u>copy</u> the original program. This copy is altered to meet the new requirements. I call this approach *metastisization*. (The somewhat perjorative connotation is intentional.) My experience has been that this is the approach most often taken in informal communities of programmers when faced by rapidly changing requirements. It is replete with problems. To name a few: New programmers usually understand only those localized portions of the programs that they are called upon to change. New documentation is rarely added, and is often incorrect. Worse yet, obsolete code and documentation often litters the derived program. A lack of comprehension of the global structure of these programs frequently leads to errors. (It is only fair to note here that some of the same sorts of comprehensibility problems can arise in massive object-oriented inheritance hierarchies.)

As such programs grow more complex, no single individual completely understands what they do. Reading and fully comprehending such code is among one of the most difficult intellectual challenges one might have the misfortune to encounter. In part for this very reason, these programs take on lives of their own. Starting over would require that somebody figure out what these programs do and rewrite them properly. Since this would usually require a large investment of the time of someone who is not a programming professional and has other things to do, these monsters live on. The behavior of these programs, and not any explicit experimental design, comes to define what the de facto designs of certain experiments are.

One of the biggest difficulties with allowing a plethora of variants on an original skeleton to proliferate is that the chore of maintaining or improving them increases n-fold, since any bug fix or improvement must be made to every copy. The version management problems, should multiple system be involved, become similarly overwhelming.

What can be done to drive stakes through the hearts of these creatures?

Let me review the lifecycle alternatives I've just given:

Disposable programs	Good for short orders, but wasteful
Parameterization	Generalization, internal complexity
Conditionalization	De facto variants, internal complexity
Metastisization	Skeleton programs, version proliferation

Brad Cox of Productivity Products has proposed the notion that object-oriented programming promotes the generation of what he calls software ICs [Cox 86]. It is interesting to observe how the evolution of IC interfaces has paralleled the paths described above in many respects. MLSI ICs were clean, simple, single purpose black boxes. New ICs were often designed by treating a previous design as a skeleton, with the features that distinguished the new design added in lieu of certain features of the base design. Todays VLSI controllers have much more complex interfaces, that provide a variety of modes (conditionalization) and internal registers (parameterization). Rather than being able to learn a single external interface, and treat the component as a black box, the designer is forced to become much more familiar with the internal structure of these new controllers.

Software vendors who wish to protect their source code have an interest in promoting frameworks as black boxes. It remains to be seen whether this is a practical approach. Vendors who retain their source code must nonetheless produce extensive documentation of the the internal structure of their components. It is interesting to note that two of the most successful object-oriented frameworks, Smalltalk-80 and MacApp, come with full source code.

Both the skeleton and VLSI controller discussions above illustrate one of the major problems with brute force generalization attempts. Attempts to encompass too much functionality in a single component can greatly increase that component's internal and external complexity. A component with an excessive number of modes will suffer from a perception of high complexity and will require a larger number of operations for initialization and configuration.

The CPL battery was in many respects a radical antithesis inspired by the woes of disposable programming and metastisization. It pushed the alternative strategies of parameterization and conditionalization to their limits, and then headed in the direction of employing a crude, brute force inheritance scheme to manage some of the subapplications as they evolved. The success of the techniques used in the CPL battery led me to investigate the use of a true object-oriented design as exemplified by the battery simulation. (Another factor was the superior facilities that object-oriented systems provide in the way of user interface components.)

Object-oriented frameworks allow a base application to serve as a dynamic skeleton for a number of derived applications. Unlike a typical skeleton program, a change to the root classes of a framework will affect all the subclasses that inherit its behavior. By allowing application specific code to reside in subclasses, frameworks encourage the designer to develop and enhance the base classes. Enhancements to a skeleton affect only those application written using it after the enhancements are made. Subsequent enhancements to a skeleton cannot affect programs previously derived from it. Frameworks used in this fashion are perhaps one of the best examples of how one should properly exploit the code sharing facilities provided by object-oriented inheritance.

The CPL battery items have repeatedly benefited from the ability that frameworks provide for sharing improvements made to their generic cores. Each time a feature is added to this core, all the battery items are able to use this feature the next time they are compiled. Managing parallel changes to every member of a large family of applications on a program by program basis can become so time consuming as to be prohibitive.

Skeletons and libraries in conventional systems can be thought of as lying at two poles along a generality axis. Frameworks bridge this gap by serving as waystations at which components can reside as they evolve from applications to framework components to library components. Note that not all components will complete such a journey. Some will never even begin. However, as a system matures, the responsible designer will make every effort to factor common code out of application classes into abstract frameworks. He or she will also take pains to find application independent components and place them in libraries.

Skeleton-based approaches have become increasingly popular over the last few years as the general level of application program complexity has increased. Programmers who several years ago might have been producing simple applications with command-based glass teletype style user interfaces are now faced with demands for complex, event-driven applications in multiprocessor distributed environments. As a result, programmers who might have written applications from first principles in the past must rely more and more on skeletal templates.

Application frameworks like the Smalltalk MVC framework and MacApp have already demonstrated the advantages of object-oriented frameworks over skeleton-based programming in the user interface domain. Enterprises like the battery simulation are a first step towards demonstrating the power of this approach outside the cozy realm of those domains well known to computer scientists.

Why Software Design is So Difficult

To understand how object-oriented frameworks might ease the plight of the software designer, it might prove useful to first examine the difficulties present in the software design process in general, with a particular focus on how traditional design practices address volatile requirements.

At the most fundamental level the reasons for the difficulty of software design are as varied as each individual set of requirements, implementation tools, and programmers. However, it should not be surprising that design is at the heart of most difficulties in software systems, since design encompasses the structural complexity of the solution that must match a given specification and the low level algorithmic idioms that must express this solution. No matter how carefully prepared or detailed a system definition and requirements specification might be, it should only specify what is to be implemented, and not how this is to be accomplished. In the end, many easily specified tasks might still be "easier said than done". Likewise, the traditional distinctions between programmers and analysts reflect a recognition that is is harder to design systems than it is to implement them.

Software design is difficult in part because software designers work in a universe of arbitrary complexity where their creations are not constrained by physical laws, the properties of materials, or, in the case of many application domains, by the history and traditions of earlier designs. Hence, the number of degrees of freedom that confront the software designer can dwarf those faced by designers in more conventional engineering disciplines.

Adding to this difficulty is the fact that only a few application domains have been studied thoroughly enough so that their design principles are well understood. Computer scientists have spent a massive number of person/years investigating the principles of compiler construction and operating system design. These efforts have allowed the construction of truly useful automatic tools, and allow the analyst working in these areas to "stand on the shoulders of giants" when confronted with design problems drawn from these domains.

However, the hapless designer confronting a problem from outside these areas is nowhere near so fortunate. The triumphs of another application domain may map poorly, if at all, to a new discipline, and the design principles of a given domain may be quite subtle. It took four centuries to discover that mimicking the birds was a bad way to design heavier than air vehicles that flew. The solution that is now commonly used, the airfoil, seems upon naive examination to be a peculiar place in the space of all possible solutions to fix as a fundamental constraint. Yet, our fleets of flying machines are now designed with just such a basic constraint. The point to this all is that for many application domains, we simply have not discovered the basic pivot points around which other constraints may be relaxed. Each domain may await solutions with the elegance of our parsing techniques, or synchronization primitives, before other design constraints can be wrestled into place. In this sense, design can be said to be "experience limited" or even "idea bound".

Another general problem with the massive constraint satisfaction problem facing the software designer is (apart from the fact that there are many more constraints than face other types of designers) the fact that design parameters vary along vague, difficult to conceptualize (let alone visualize), frequently non-scalar dimensions. A bridge designer can gauge the effect of adding additional concrete to a roadbed in terms of weight, volume, and the like. However, the effects of manipulating software design parameters (coupling, cohesion, number of modules, module size, intermodule protocols) and the effect of changes to these on functionality, reliability, generalizability, reusability, etc. are difficult (at best) to accurately estimate.

One of the worst effects of this state of affairs is that it is not always easy to tell if a specification is unreasonable. Considers the example of constructing an automobile that can fly using parts found at local shopping malls. I will submit that some of the problems that confront software designers are in a very real practical sense every bit as outlandish as the task given above. The specification might be quite clear, and the implementation materials and tools well understood, and the task might well be theoretically possible. Yet, in the absence of experience to guide the design process, a designer might not be able to distinguish between the software equivalents of the merely difficult task (building an automobile from parts) and the much more difficult task of allowing it to to meet all the previous constraints on automobiles and fly as well. If the solution domain is such that there is no major axis of impracticality, but some number of axes along which a design must be optimized, the task can become quite daunting.

How can these sorts of problems be addressed? Clearly we must give the designer the means to limit the huge number of design decision he or she must make, and to identify those aspects of the design that have a major impact on the suitability of the resulting product to the task for which it is being designed.

How might the number of design decision be limited? One approach is to allow the designer to refer to an existing body of work that meets similar constraints, if not perfectly, then at least in an aesthetically reasonable manner. Kristina Hooper [Hooper 86] points out that software design often has as much in common with disciplines like architecture as it does with disciplines more traditionally associated with it (such as engineering, mathematics, and physics). From this perspective, questions of aesthetics, style and experience play a much more central role in the design process, and judgement as well as empirical constraints may legitimately serve as constraints on the design process. Hooper points out that in software design, as in architecture, fads and even "schools" of design orthodoxy can rise and fall. Of course, previous "conventional" design experience often comprises a rich lode material for the designer as well. However, in those domains where previous principles are not well established, intangibles like experience, judgement and aesthetics can reasonable be brought into play. It is probably also the case (to paraphrase [Kernighan 76]) that with design as well as with programming, good practice is developed as much from looking at actual working systems as from "abstract sermons". Disciplines like architecture and engineering can teach us much about the value of case studies. Object-oriented programming, with its emphasis on reading code, provides more opportunity for this that most other systems.

Fred Brooks [Brooks 87] makes a similar point in his paper "No Silver Bullet: Essence and Accident in Software Engineering". He states: "We can get good designs by following good design practices instead of poor ones. Good design practice can be taught." However, he goes on to say: "Whereas the difference between poor designs and good ones may lie in the soundness of the design method, the difference between good designs and great ones surely does not. Great designs come from great designers. Software construction is a *creative* process."

A second point at which to limit design chaos is at the interface between the software system being designed and the substrate upon which it is being built. Hence, the programming environment under which a design is undertaken can do much to limit the number of arbitrary choices that a designer must make. One way it can do so is by providing a large library of fixed, well understood, reusable components (see [Harrison 86]). A knowledge-based system can even apply design schemas to coding situations into which template idioms can be fit (see for instance [Lubars 86]). In either case, having a set of "canned" idioms serve as low level reference points in the design constraint satisfaction process is of value to both human and automatic "designers". Most designers recognize that working up from a set of fixed low level components brings many advantages to an ostensibly "top-down" design exercise. It is frequently observed that design is an iterative process in which fixed points at either end of the design "space" reach in from both the top-down and the bottom-up to attempt to fix additional decisions.

A third aid to the designer is the set of rules and guidelines that have been developed to guide the design process itself. Our variations on the ancient edict to "divide and conquer" and on the notion of "need to know" help guide the designer to limit the internal complexity of his or her designs. Notions like stepwise refinement, hierarchical decomposition, and information hiding guide the designer to keep the intermediate nodes in the design graph intellectually manageable. Admonishments to keep cohesion high and coupling low (see [Pressman 82]) give the designer goals to try to achieve as he or she wades through the muck between the desires expressed in the specifications and the reality of what problems there are already solutions at hand for. However, despite the best of intentions, sometimes the problem has ideas of its own. Only the most hardened theoretician could believe that for every problem, no matter how inherently complex, sloppy and idiosyncratic, there must be one or more elegant, tidy solutions.

Programming tools that help express complex constraints among system components can help somewhat as the going gets thicker. Alan Borning's ThingLab [Borning 79] and other work on constraint satisfaction systems shows that more powerful systems and notations can aid the designer in dealing with complex problems. Borning also cites Simon's observations about "nearly decomposable systems" that can almost, but not quite, be composed into cohesive units. Borning's system displays problem solutions built from objects that are more or less cohesive units ("wholes") in themselves, while they at the same time comprise the components ("parts") of higher level "wholes". Such part/whole type organizations are frequently seen in both artificial and natural systems.

For certain classes of problems, very high level languages, or declarative or knowledge-based languages can force the designer down paths that are known to have a high likelihood of leading to success.

Tools that help the designer visualize the structure of the system he is designing can aid the design process as well. Systems like SADT [Ross 85] provide facilities for divining and visualizing the structures of existing systems, including ones under construction.

Still, the situation must at times seem fairly dismal to the large system designer. He or she works in a realm where the normal physical bounds on system complexity do not exist, and the roots of complexity are poorly understood. The benefits of the insights of Eli Whitney and Henry Ford (i.e. standard interchangeable, reusable parts) are still largely unavailable to software designers in most application domains. (This lack of standardization is in part the fault of a tension between the desire for generality, and a desire to achieve simplicity in individual designs. More on this later.) Programming tools can help at the lowest levels, but as the size of a system increases, the entire effort can become increasingly design intensive. Finally, good tools to help the designer understand the structure of existing systems are only beginning to appear.

All of this might seem bad enough, but wait, it gets worse...

Designing in the Presence of Volatile Requirements

Software design for all but the most trivial applications is a difficult process under even the best of circumstances. In the traditional waterfall software lifecycle model, software design is performed once and for all after initial system definition and software requirements assessments have been made. Forcing the designer to confront the probability that system requirements will change throughout the lifetime of the software product adds a new dimension of complexity to the design problem. Few treatments of the software design process address this possibility. Changing requirements are nonetheless a fact of life in most application domains.

Dave Brown, in an article entitled "Why We did So Badly in the Design Phase" [Brown 84] points out that by putting all the design effort up front, we run the risk that that effort will be deemed to be difficult to abandon, even when a better solution is identified later in the process of product development. In the horror story he cites, a design group was reluctant to give up the effort it had put into a cumbersome design, simply because of how much effort had already been expended in developing and documenting that design. In the end, a fresh team member shows the original team the error of its ways.

One could construe this as merely an example of Brook's [Brooks 75] admonition to "plan to throw out the first one. You will anyway." I think the example might serve to make a stronger point than that. That is, that since change is inevitable, we should anticipate it and plan for it.

There are many reasons why the designer often has the requirements rug pulled from beneath him. Sometimes, change requests come from users who do not fully comprehend the impact that software changes might have on the structure of a software product. This problem is addressed below. Certainly one way of addressing the problem of volatile requirements is to attempt more vigorously to prohibit them. My feeling is that, as helpful as this might be, it is simply not realistic to expect that this is possible in many (perhaps most) cases. I would argue instead that we must accept volatile requirements as inevitable, and develop design techniques and tools to cope with them.

One of the more difficult problems facing the software designer is the myth of software malleability. This myth holds that since a given system is implemented in software, it must be arbitrarily easy to change. This belief is common among users and customers with no or (worse yet) modest amounts of programming experience. It is reinforced by experience with small programs that can lead one to generalize incorrectly from the experience one perceives of the ease with which one can make a minor changes. The naive seem to think that this phenomenon scales directly in such a way so as to make arbitrary changes to large systems "trivial".

The truth is that in many instances the structural edifice that constitutes the software architecture of a system is its most "rigid" part. As systems grow in complexity, the lattice of architectural dependencies between its constituent parts can become more and more dense. Such patterns of coupling among the components of a system reflect real complexity in the architecture of the system that result directly from the complexity inherent in the original requirements and the design choices that were made in divining particular solutions to the problems posed by those requirements. This perspective makes it easy to see why sometimes emulations are the most effective way to keep existing useful systems in the field working. In these instances the observation that "software is hard, and hardware is soft" is appropriate. The person charged with maintaining the system in these cases may have correctly recognized that the instruction set level interfaces was the most structurally simple place in the system to intervene.

Another source of the impression that software should be more malleable than it is is a failure to recognize that the order in which design decisions are made affects other decisions in ways that are difficult to undo. Design in practice is neither truly top-down or bottom up, but it is a hierarchical process. At the top of the hierarchy are the broad outlines of the product being designed. At the bottom are those things we already know how to do that we recognize as likely to play a role in the solution to the problem at hand. The designer fixes decisions at a given level by considering the impact they will have on their ancestors, neighbors, and descendents in the design hierarchy. At each level, decision making is influenced by both top-down and bottom-up feedback. The designer attempts to come to an optimal solution to the issues present at each level of the hierarchy before turning his or her attention to the next level. As decisions at each level are fixed, they become fixed constraints around which other constraints are relaxed. Attempts to violate the structure of a system so constructed can be very disruptive. To give an architectural analogy, it is much more appropriate to make changes to the wiring plan of a building <u>after</u> the floor plan is determined, and <u>before</u> the wallboard is up. Considering this level of detail too early can be counter-productive, since the wall locations have yet to be determined. Later changes can require radical backtracking (such as tearing down the wallboard).

Incorrect perceptions of software malleability are but one phenomenon that drives change and evolution in software systems. Changing requirements, the need to add additional capacity, the need to repair existing problems, and increased user sophistication are other factors that can foster system change. This list is by no means exhaustive. I'd like to concentrate here on externally initiated change requests, that result in a system's having to meet what are in essence dynamic "moving target" requirements.

The requirement that a system designer somehow design to facilitate change adds a whole new dimension of difficulty to the design process. The easiest position to take is to ignore all but the current system requirements. This is only done, however, at the expense of avoiding obvious opportunities to generalize system components in such a way as to anticipate future requirements. This tension between generality and simplicity (and specificity) is a pervasive problem facing the designer. Does the designer build a simple component with poor generality but (perhaps) a simple, cohesive interface, or a more general, but (perhaps) more complex component that might require more parameters and a more complicated interface? How does the designer gauge the potential for possible reuse, especially in the face of changing requirements? To put it another way, does one narrowly solve only the problem at hand, or does one spend considerable additional effort to build general components that solve a more general range of problems, some of which one might (or might not ever!) have to address at some future date? If one attempts to do this, how does one know that he or she has anticipated the right problems? Where does one draw the line?

Assume that any component lies somewhere along a generality continuum. At one end, there are actions, such as string manipulation, basic display primitives and the like, that obviously are sufficiently context independent, and of sufficient general utility to justify the effort needed to make them general tools. At the other end of this continuum are those portions of any programming task that are manifestly specific to the application itself. In between is a broad gray area that up until now has been claimed by default by the application specific end of the continuum. It is in this area that I think application frameworks can help the designer the most.

Recall that frameworks can allow the abstract designs present both in specific applications and libraries to be customized, extended, and reused. Since a framework can encompass a family of diverging, related components, less pressure is placed on the designer to initially anticipate all the demands that might be placed on a given component. The designer can address the problem at hand, knowing that the resulting component can serve as the root of a framework as related problems arise. Also, the designer can seize opportunities to exploit potential generality, since a general component can be recustomized to meet specific application demands that might if necessary.

Recall, too, that as such frameworks emerge, any enhancements made to their core components will be shared by other elements of the framework. In conventional systems the only way to accommodate divergence in an evolving family of applications is to maintain separate sources, and share only those application independent components that are characteristic of conventional libraries. This proliferation of source files and versions can be a nightmare for the designer entrusted with maintaining them.

I believe that the use of object-oriented application frameworks can help the designer to design in such a way as to facilitate change. Such frameworks can also foster the sharing of generic skeletons and somewhat application specific components, and can help the designer avoid the haphazard, patchwork organization frequently seen in evolving systems during the maintenance phase of the software lifecycle.

A final problem confronting the designer is designing systems to be safe and reliable. As if designing the system to work the way it is supposed to is not enough, the designer is confronted as well with the responsibility for making sure that his or her system performs reasonably in the face of input or uses for which it was not intended. Here too, the process of making a system more robust is much more complex than that facing designers in more conventional disciplines. An aircraft engineer can increase safety by increasing design margins. If only typing each line of code five times would make a module five times more reliable.

One way to aid the designer in dealing with these problems is to provide a greater armamentarium of off-the-shelf, proven components. Such components can be more reliable since they will often have been tested and proven in previous applications. Since it is known that components in such a library may be frequently reused, the software engineer can justify lavishing more attention and resources on them that might be justified were they being used only one. The potential range of application of such components can be substantially increased using schemes like the application framework approach.

The plight of the designer is daunting, but not hopeless. Despite the many perils that face the software designer, programs that appear to work still manage to appear. We are only beginning to understand the real complexity of these undertakings, and construct tools and techniques to come to grip with them.

Designing to Facilitate Change

Over the last twenty years, computer scientists have made impressive progress in understanding how to design and implement computer programs given a fixed set of requirements. Much less is known, however, about how to design systems to facilitate their orderly evolution. Designing to facilitate change is a much more demanding task than designing simply to meet fixed requirements. Managing systems in the presence of volatile, changing, highly dynamic requirements can be a daunting task. Lifecycles that are characterized by a steady infusion of new requirements are qualitatively different from the traditional "waterfall" lifecycles.

[Booch 86] has discussed the use of object-oriented development techniques as a partial lifecycle method. [Jacobson 86] discusses how object-oriented approaches can support change in large realtime systems.

It is far more difficult to design a framework that attempts to accommodate future expansion and extension requirements than it is to merely meet the requirements at hand. How does one trade off the simplicity of solving only the current problem with the potential benefits of designing more general components? There is, (to paraphrase Randall Smith), a tension between specificity and generality.

Kent Beck [O'Shea, 1986] claims that turning solutions to specific problems into generic solutions is a difficult challenge facing system designers using any methodology. To quote: "Even our researchers who use Smalltalk every day do not often come up with generally useful abstractions from the code they use to solve problems. Useful abstractions are usually created by programmers with an obsession for simplicity, who are willing to rewrite code several times to produce easy-to-understand and easy-to-specialize classes." Later he states: "Decomposing problems and procedures is recognized as a difficult problem, and elaborate methodologies have been developed to help programmers in this process. Programmers who can go a set further and make their procedural solutions to a particular problem into a generic library are rare and valuable."

Useful abstractions are (more often than not) designed from the bottom up, and not from the top down. We create new general components by solving specific problems, and then recognizing that our solutions have potentially broader applicability.

Specificity vs. Generality

It would seem that the point at which it pays to design a general rather than a task specific component is different in an object-oriented environment than in a conventional environment. The main reason is that polymorphism and inheritance increases the likelihood that any given component might itself be usable in a wide range of contexts, and that other components can be reused within it.

However, generalization has a cost. Attempting to design a general component requires a great deal more thought and effort than designing a narrow, single purpose component. Furthermore, such efforts are not at all certain to succeed. It is not at all obvious how many specific problems will be special cases of a particular general solution. It may be the case that a component can be made more general only at the cost of increased internal or external complexity. (For instance, it should be clear that any given component could be made to solve two problems with the addition of an external "switch" and a number of internal tests on this switch.) In all but the most obvious cases, it may not be easy to determine how useful it is to make a given component more general. The designer will at times engage in the speculative design of general purpose components that solve problems that may never arise.

A general design can be like a swiss army knife. The danger is that a complex tool may perform many tasks, but none well. As with the swiss army knife, this approach may sometimes be just what a given job requires, and may be inadequate in other cases.

The goal of the conscientious designer is to find simple solutions that encompass not only the problem at hand, but a wide range of other as well. These are the holes-in-one that more than make up for the many blind alleys that the designer may have explored.

As has been noted before, the use of inheritance hierarchies can allow an evolving set of components to occupy intermediate states between being completely specific to one problem and being largely context independent. The use of subclassing to allow a given core to solve several related problems provides an alternative to conditionalization, and can allow the outlines of an abstract superclass to emerge as a given set of classes is reapplied to successive sets of requirements.

Inheritance can help the system designer avoid the loss of generality that can occur during mid-life phases of the software lifecycle when the pressures to push a clean design beyond its original specifications inevitably arise. A well designed software component should have but a single purpose (that is, it should exhibit a high degree of cohesion). (See for instance [Pressman 82].) However, it is often easier to pervert a clean single purpose component to make it serve multiple roles than it is to build new components from the ground up. Thus, a previously general component may have task specific features added to it.

The pressure to do this may be particularly hard to resist when a component that was conceived of by the designer as truly general is currently being used for only one or two purposes. Short sighted managers are often willing to trade long term, hypothetical generality for such short term gains. The wisdom of such trade-offs can only be gauged in hindsight. The use of a framework-based approach can reduce the pressure to lose generality by allowing a generic core component to retain its generality. Application specific features are implemented in subclasses using inheritance.

In this way, object-oriented systems provide a continuum along which system components may evolve from specificity to generality. A given class hierarchy containing an entourage of task specific subclasses provides an intermediate stage that is much harder to manage using conventional approaches.

The ability to allow a family of applications to be managed as they evolve and diverge is very valuable in environments characterized by highly dynamic, rapidly changing (even fickle) requirements. Laboratory research applications are, by the very nature of research itself, such environments.

Objects, Evolution, and Maintenance

It is not difficult to argue that evolutionary lifecycles are the rule rather than the exception in practice. Sofware lifecycle researchers are increasingly recognizing the distinctive problems that must be addressed during a software product's evolutionary phase. (See for instance, [Lehman 80], and [Boehm 85]. Fairley observes that "the distribution of effort between development and maintenance has been variously reported as 40/60, 30/70, and even 10/90" [Fairley 85]. Pressman [Pressman 82] quotes figures (from [Lientz 80]) that identify four software maintenance categories: *perfective, corrective, preventative, and adaptive.* Corrective maintenance is the process of diagnosing and correcting errors (i.e. the bug fixing that is usually thought of as the maintenance phase of the software lifecycle. Adaptive maintenance consists of those activities that are needed to properly integrate a software product with new hardware, peripherals, etc. Perfective maintenance is required when a software product is successful. As such a product is used, pressure is brought to bear on the developers to enhance and extend the functionality of that product. Preventative maintenance activity occurs when software is changed to improve future maintainability or reliability or to provide a basis for future enhancements. [Lientz 80] reported in a study of 487 software development organizations that a typical distribution of these activities was:

Corrective	21%
Adaptive	25%
Perfective	50%
Others	4%

This observation and the observation that 50% of maintenance activity is "perfective" would seem to support the contention that an evolutionary phase is a inevitable (and quite important) phase in the lifecycle of a successful software product.

The term "maintenance" phase has a certain perjorative quality about it. It has come to be associated with the drudgery of dealing with, debugging, and mopping up after other people's code. In large organizations, maintenance tasks are often assigned to junior programmers. After having spent some time doing maintenance tasks, programmers can then graduate to more glamorous development work.

The term maintenance implies that this part of the software lifecycle consists of the software equivalent of checking the oil and changing the filters. I would contend instead that it is at this stage in the evolution of a software system that many of the most important and difficult design challenges are encountered.

In many respects, maintenance is a more demanding task than development. The maintainer must infer a complex design, frequently given inadequate or absent internal documentation, and make modifications that might cut across the grain of the existing design.

It is not surprising, then, that senior people in an organization might reserve for themselves the relatively pleasant, rewarding tasks of designing and implementing clean new systems from initial requirement sets, and leave to others the task of molding existing systems to changing requirements. This is because doing a new design for any one set of requirements is a relatively simple, high profile task, while adapting another programmer's existing system to requirements that contradict those of an original design can be a difficult, tedious, and relatively thankless job.

This is an ironic corollary to the Peter Principle. The person best qualified to maintain a system component, its original designer, is promoted as a result of the success of this design effort to a position where he or she is no longer responsible for the component's further evolution.

The design clashes that occur during the maintenance/evolutionary phase of the software lifecycle that normally confound the original design can often be minimized by an application framework. Because application frameworks provide a middle ground between generalizing a single application to meet contradictory requirements (conditionalization) and spawning new versions (metastisization) they can provide a path by which sub-versions of the original application can gradually diverge from the original design.

By promoting the sharing, reuse and understanding of other people's code, object-oriented environments can help enhance the appeal of perfecting and adapting existing systems over that of designing new systems.

Reuse vs. Reinvention

There are limits to how complex a system a given person can construct, manage, and comprehend. No (sane) person would dream of attempting to single-handedly engineer an entire automobile, or a Saturn-V. Yet, many a contemporary software engineer harbors (with out realizing it, in many cases) equally outlandish expectations about what it might take to construct certain software systems from first principles. Faced with reusing the work of others, many would elect to reimplement a given component instead. An aversion to code "not invented here" or even "not invented by me" is quite common in programming circles. Why are programmers so often inclined to rewrite code rather than reuse code?

Often the fig leaf of "source control" is cited as a reason why existing products must be reimplemented. Certainly ego is often the reason that code is reinvented. There are those who aspire to reimplement the world, with their name at the top of each module.

There are more benign reasons as well. One is the difficulty of reconstructing the assumptions and implicit design principles that underlie a given module by reading the code for it. The difficulty of comprehending what a given module does by merely reading its code is, I think, one of the most underestimated things in all of computer science. Also, as most programmers know, the state of the internal documentation in most production modules is typically quite poor (to say the least). (To jump ahead to learnability, the question is not why Smalltalk is hard to read, it is why we compare a programming style that requires extensive reading with one that does not.) In most environments, programmers are seldom called upon to (or seldom undertake to, in any case) understand much more than very localized portions of existing systems. Most such encounters take place during the so-called maintenance phase of a given product's lifecycle.

Hence, one of the major motivations for rewriting existing code is to ensure than the design assumptions and structure of that module are completely understood.

There seem to be but two ways out of this conundrum. One is that reusable components must be written using a level of abstraction that makes them easy to comprehend. The second must be that the effort to learn the internals of a given component must be rewarded by a high level of generality and reusability. That is, either we must only learn a component's external interface, or learning the internals must be worth it in terms of reuse potential. (One path leads to data abstraction and black box frameworks, the other to extensible white box inheritance frameworks.)

If a component is likely to be reused, then a greater level of effort to make it more general can be justified. Object-oriented languages can have just this impact. Polymorphism increases the likelihood that a new component will behave correctly right away in a number of existing contexts, and that that component will be able to operate on a potentially wide range of different objects. The judicious use of inheritance can allow the construction of very general abstract cores, (i.e. frameworks), that can then be extended and specialized as specific requirements dictate.

Note that frameworks occupy a position in between application code and library components in terms of generality. A component can be reused as a black box, the behavior of which is defined by its protocol (or signature). (Re)using a framework, on the other hand, requires that existing code be treated as a <u>white box</u>. Subclassing the components that comprise a framework requires that the programmer have a detailed knowledge of the internal structures and mechanisms in order to choose judicious overrides to enable the framework to implement the problem being addressed. Even in those cases (for example MacApp) where the override points are well specified, extending a framework might be said to require a "gray box" understanding of the framework.

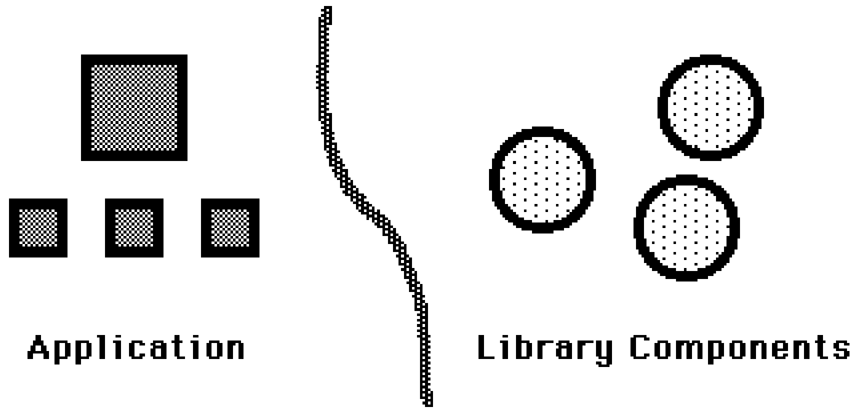
William Cook, in a forthcoming paper on the denotational semantics of inheritance [Cook 87] makes a similar observation. He claims that an object has two interfaces, an outer interface that is seen by code that uses the object, and an inner interface seen only by its subclasses.

Frameworks provide a way of reusing the sort of code that is resistant to more conventional reuse attempts. Object-oriented libraries provide reuse capabilities for application independent components that are analogous to (though considerably more powerful than) those seen in the utility and support libraries used in conventional environments. However, it is very difficult to use conventional systems to build general abstract algorithms that can be used in a wide range of different contexts. Hence, we can reuse whatever application independent components we might generate in the course of generating an application rather easily, but reusing the edifice that ties the components together so that they solve a problem of interest is usually possible only by physically copying the application. (This is the skeleton program approach.)

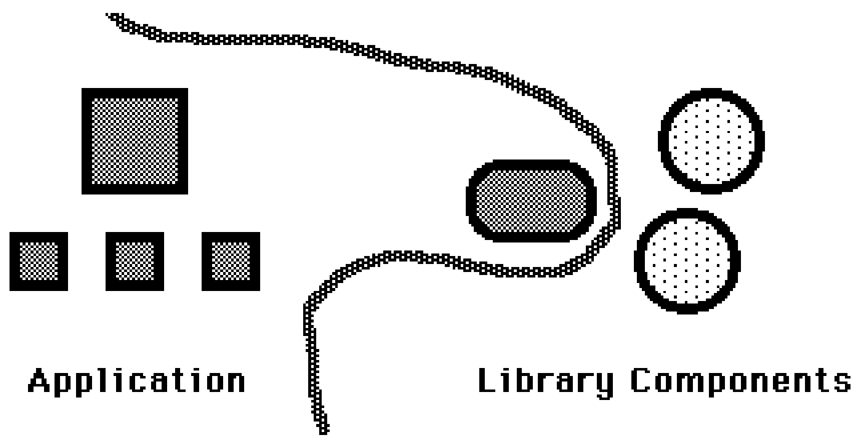
Approaches like using procedure and function parameters, and Ada generic modules address these concerns to some extent, but with nowhere near the generality of an object-oriented framework approach. One can make the case that the use of these techniques is a sort of brute force attempt to reap the benefits of object-oriented polymorphism and inheritance using conventional programming tools.

Frameworks and the Software Lifecycle

This section depicts the relationships among applications, frameworks, and library components. It attempts to illustrate how the use of frameworks can bridge the gulf that traditionally exists between application specific and application independent code.

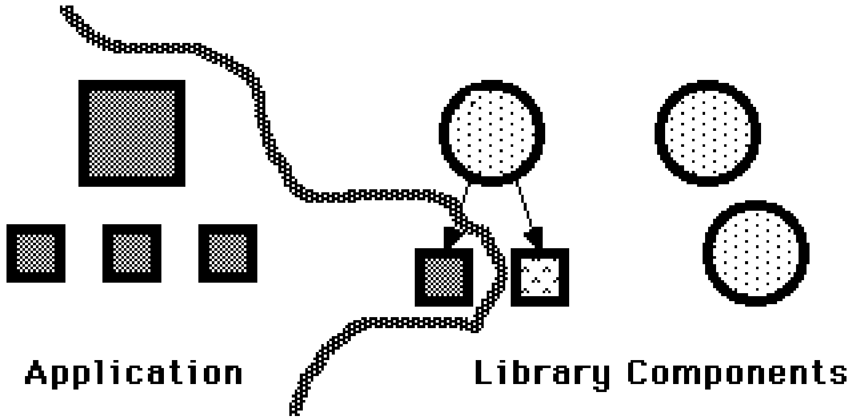


The figure above shows an application program surrounded by a set of application specific subcomponents (the squares). The circles at the right represent application independent library components. The gray line shows the boundary between application specific and application independent code.

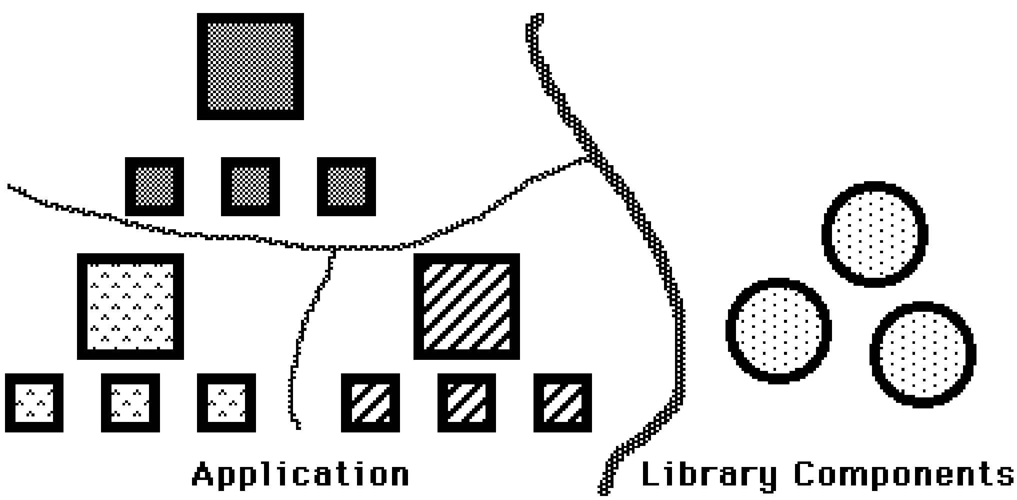


The figure above illustrates a phenomenon that can occur during maintenance. Expediency can force maintainers to modify a previously general component to meet the needs of a specific application. A loss of generality ensues. This can be seen in the way that the boundary between application independent and application specific code has shifted to encroach on what was previously application independent territory. In practice, this phenomenon is most frequently exhibited in somewhat application domain specific components that were originally designed for the given application, with an eye towards reuse. The pressures of the maintenance phase can result in the benefits of such foresight being squandered.

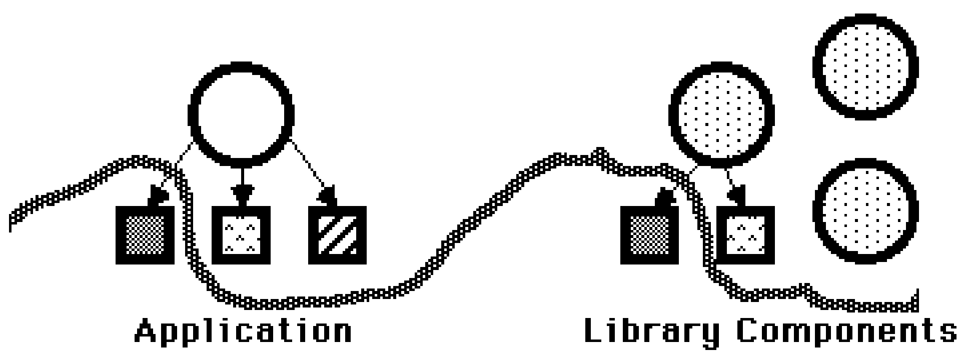
A variation on this phenomenon is when the previously general component is used as a skeleton for the new component. This leads to a proliferation of components derived from a given skeleton.



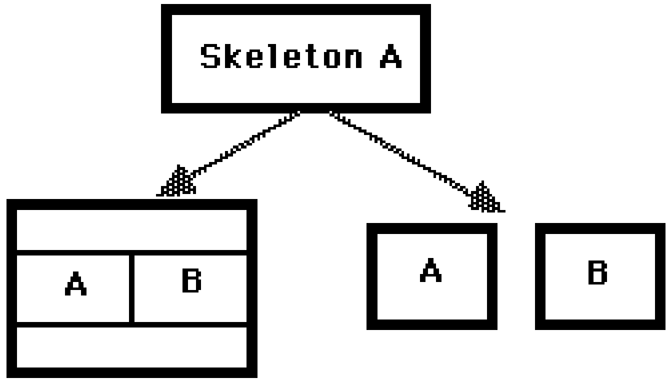
If a library component is used as the root of a component framework, the problems discussed above can be avoided. The figure above shows how the specializations needed to the original component can be embodied in a subclass of that component. This subclass will share most of the original components code, and inherit any subsequent changes made to the original component. Note how the boundary between application specific and application independent code has been gerrymandered over into the library component's realm to encompass the application specific subclass.



The figure above depicts two applications (the lower rectangles) derived from an original application that has served as a skeleton for them. The large rectangles represent the mainline application code. The smaller rectangles are application specific subroutines. The large circles are library components shared by all three programs. Though all the applications share the same library components, they share no application specific code. Hence, any application specific changes required by all the programs must be made separately to each. This barrier between the original skeleton application and the two derived applications is represented by the thin gray line. This process becomes increasing difficult as these applications evolve and diverge, since what should be common code may not be maintained as such in any orderly fashion.



This figure shows how an application like the one from the previous figure might be organized using a framework-based approach. On the application side, each application is a subclass (the small rectangles) derived from what is now a general core on the application specific side (the plain circle) and the library components. Notice how the gray boundary line now encloses only the two gray subclasses. The original application is now itself serving as the root of a framework. Since this inheritance framework is itself shared, it is drawn as a circle. Changes made to this common core will be reflected immediately in each derived application. Each application is comprised only of that code that distinguishes it from the roots of the frameworks, and is likely to be much smaller and easier to comprehend than the corresponding skeleton program. The identities of the application specific subroutines (the small boxes) will not necessarily remain distinct when a set of skeletal applications is transformed into a framework. (They do not in this example.)



This figure shows a dilemma that faces programmers using conventional tools when they are faced with changing requirements that are at odds with an applications original specifications. They may either create a conditionalized variant that encompasses the functionality of both sets of specifications, or use the original application as a skeleton for a new application. The former approach increases the internal complexity of an application, since it becomes filled with case-like statements that test variant tag fields. On the plus side, each variant will continue to retain all the functionality of the original application as both evolve. The latter approach introduces all the problems associated with managing a family of skeletal applications.

A framework-based approach combines the power of the first approach with the code level simplicity of the second approach. Since message sending will in effect perform the dispatches on the variant tags implicitly, the subclasses can retain the structural clarity of single purpose components while retaining the shared capabilities of their ancestors.

Programming in the Smalltalk-80 Environment

Developing a system using the Smalltalk-80 environment is distinctly different sort of enterprise from doing so using a conventional environment. The most obvious difference is that from the onset, one spends ones time reading (other people's) existing code, rather than writing (or reinventing) new code. This is because in order to exploit the vast body of existing code that one extends (subclasses) in order to create new, application specific classes, one must understand the capabilities of the classes that already exist.

The rich vocabulary of preexisting classes is what gives object-oriented systems much of their power. It is the need to learn the capabilities of the classes in these libraries that is at the root of some of the "learnability" problems reported with some object-oriented systems. In any case, it is the presence of this huge library that results in the need to spend time reading code rather than writing it.

The Learnability Gap

A drawback cited by some to the use of object-oriented environments is that programmers find them difficult to learn [O'Shea 86]. My experience using the Smalltalk-80 environment has been that it does indeed take a great deal of time to learn it well. The learnability gap is real. However, there are reasonable explanations for it.

By far the most significant factor affecting the relative learning times in systems like Smalltalk is the need to to master an enormous library of methods before one can begin to intelligently use the system. A conscientious newcomer to Smalltalk may spend almost all of his or her time reading existing code. The same programmer using a language like C would probably begin by reimplementing the very primitive operations that the Smalltalk programmer is searching the existing code for. So in one sense, comparing the Smalltalk environment with the C language in terms of learnability is to compare apples and oranges, it is the Smalltalk language itself (which is quite small) that should be compared with languages like C. A much fairer comparison might be to compare the learning time for the Smalltalk system with that of the Unix environment.

The notion that long learning curves are caused by the need to master a large library of components is borne out by the experiences reported by programmers attempting to write applications using the Apple Macintosh Toolbox [Apple 85]. This subroutine library is not object-oriented (though it was heavily influenced both by Smalltalk, and an earlier objected oriented library, the Lisa Toolkit [Apple 84]). The sheer size of this library, and the requirement that a substantial subset of it be mastered before a simple application could be constructed has evidently lead to learnability difficulties that are quite similar to those reported with object-oriented systems. In fact, the addition of an object-oriented superstructure, MacApp [Apple 86], is reported to greatly facilitate the generation of Macintosh code [Schmucker 86].

Another factor that comes into play is the emphasis that is placed on the support of window-based interactive graphical user interfaces in systems like Smalltalk. Comparing the difficulty of developing an MVC-based graphical application with the effort involved in writing a "Hello world" program may again be comparing apples and oranges. In this respect, the comparisons among Smalltalk, the Macintosh Toolbox, and MacApp seem quite apt.

Hence, the sheer size of the method library under Smalltalk, is, in my opinion, the greatest contributor to the relatively long time one needs to master it. This is consistent with what others have reported [O'Shea 86].

There are other factors at work here. The requirement that a programmer actually read another programmer's code is one that is met by a surprising degree of resistance in some circles. My feeling is that the difficulty of reading and comprehending someone else's code is one of CS's "dirty little secrets". The browsing tools under Smalltalk-80 are the best tools I have seen for gleaning information from source code. However, even better tools and browsers would be helpful. (See [Borning and O'Shea 86].

Another problem is that much of the Smalltalk system is underdocumented. The absence of good descriptions of the rationale for much of the system's structure, especially in much of the graphics and MVC code, is particularly troublesome. However, even if the final Smaltalk-80 book were to exist, better facilities than are currently available for managing system documentation within the system itself could prove very useful.

An additional problem with learning Smalltalk-80 is that as much as polymorphism and inheritance aid the implementor, they can hinder the reader.

Polymorphism makes it difficult for the reader to determine the flow of control. (Ironically, this information may have been known to the author of the original method. In these cases, the availability of a type system might enhance readability by increasing the reader's certainty of his or her knowledge of the flow of control.)

The readability problems associated with inheritance seem as if they could be ameliorated through the use of better browsers. See, for instance [Goldberg 82] and [Foote 87].

Another approach that might aid comprehensibility might be the addition of a module mechanism [Borning and O'Shea 86] or the formalization of protocols.

Smalltalk and Realtime Systems

It may seem peculiar to use Smalltalk-80 system to investigate realtime laboratory programming problems. After all, realtime systems must operate under tight, demanding performance constraints, and Smalltalk has a reputation for being anything but fast. However, a need for high performance is but one of the characteristics that distinguishes the laboratory programming problems from more traditional programming problems. Among the other demands that the laboratory application domain places on the software designer are a need for highly interactive graphical user interfaces, and a need to respond to highly dynamic software requirements (i.e. requirements that are changing frequently).

The laboratory environment shares with traditional application domains the need for good editing, debugging, and code management tools.

It was the observation that the strengths of an object-oriented programming environment like Smalltalk-80 might be well matched to <u>many</u> of the needs of the laboratory programming domain that motivated this project.

Note that to achieve accurate realtime event handling, it is essential that input events be accurately timestamped and that output event initiation be accurately scheduled. If these two requirements can be met in such a way as to maintain accurate system timing, the software that schedules and consumes these events can be slow and sloppy from a timing standpoint, as long as overall throughput constraints are met (and the global timebase is precisely managed). In an object-oriented system, this might mean that a foreground process that allows tightly timed events to be produced and consumed by slower code could be used to achieve adequate realtime performance. Such table-driven scheduling and timestamping techniques have been used in more conventional situations where realtime events must be managed by slow processes.

A number of approaches might be taken to restore some of the performance that is lost when a system like Smalltalk is employed. Among these are:

The use of hybrid object-oriented languages
The use of custom primitives
The use of a typed, optimized Smalltalk system
Increased exploitation of multiprocessing and multitasking

Hybrid object-oriented languages like C++ [Stroustrup 86], Classcal, Objective-C, and Object Pascal [Schmucker 86] can allow performance levels approaching those of the base languages (in these cases C and Pascal) to be attained. In certain cases, hybrid object-oriented implementations have exceeded the performance of their counterparts written in the base languages. These improvements resulted because the use of an object-oriented framework allowed superior library routines to replace slower application routines [Schmucker 86].

Another approach one can take is to use efficient primitives to implement operations for which performance is critical. Such primitives might be written in fast, conventional languages like C or assembly language. This, of course, is what Smalltalk itself does for operations like graphical bit block transfers.

A third approach might be to use a highly optimized, "typed" variant of Smalltalk [Johnson 87]. This approach, in effect, turns Smalltalk itself into a hybrid language. In those contexts where polymorphism can be reduced to static overloading, the full efficiency of compile-time typing can be exploited.

It might be possible as well to exploit hardware parallelism to restore some of the performance lost to runtime polymorphism. Such improvements might result from parallel garbage collection schemes, or the delegation of time-critical tasks to dedicated processors.

Also, the sorts of functional separations made possible by language level multitasking (see the StimulusGenerator discussion in Chapter III) might facilitate the generation of more efficient runtime code, as well as battery modularity and reusability.

The StimulusGenerator hierarchy is a portion of the battery simulation that was formerly embedded in the BatteryItem hierarchy that has, under Smalltalk, evolved into a separate part of the overall battery framework. Indeed, the StimulusGenerator hierarchy is in some respects still in transition between membership in the original battery hierarchy and clean isolation from it. The degree of coupling seen in the interface between each battery item and these classes is still somewhat high.

One very intriguing route by which this coupling might be reduced is via the introduction of parallelism. Under such an approach, stimulus generators would run in parallel with battery items, with both tied to the same underlying timebase. Such an approach would simplify both the battery item stimulus coordination code, and the stimulus sequencing code in this stimulus generators. Hence, the use of parallelism could greatly increase the generality and reusability of components of both hierarchies.

Realtime application programs frequently exhibit a structure in which the mainline code must explicitly schedule sequences of otherwise unrelated events. The use of parallel processes can free the designer from this temporal yoke.

Starting with a "Real" System

By building a system that was based one built from real, production requirements, I believe I was able to avoid the artificial simplicity seen in many academic demonstration systems. It is easy to specify a set of requirements that in effect are a "straw man" waiting to be dealt with by the techniques being demonstrated.

This approach is not without its problems. In working in an area that has not been well studied, it is more difficult to evaluate the conclusions that one reaches about the effectiveness of techniques for coping with problems one must address in that domain. One also runs the risk that superfluous detail and breadth may confound one's ability to adequately focus on areas of legitimate academic interest. Both these issues arose to some extent during the Battery simulation effort.

On the whole, I feel that having used a system like the Battery as the basis for a project such as this one has been a sound idea. The combination of issues it raised, such as volatile requirements, data management, data flow problems, parameter management, and realtime device simulation, is quite unorthodox. Computer science research has a tendency towards parochialism. By addressing problems outside the veins we have traditionally mined, we increase the potential that we will gain insights of value not only to the new domains investigated, but to computer science in general as well.

Lisp, Simula, and Uniform Access

One of the most difficult design bottlenecks in the battery simulation effort was the design of the parameter access and data management mechanisms. These difficulties arose because Smalltalk's dual heritage from Lisp and Simula made it more difficult that I would have expected to design the sorts of active data structures I needed.

The problems arose for the following reasons. I wanted battery items to be able to access their user alterable parameter values in an efficient fashion. At the same time, I wanted to be able to write a general utility that would allow users to inspect and alter the parameters of <u>any</u> program, with facilities such as help and error checking. This utility would have resembled the CPL battery parameter browser. Also, it was my goal to use the objects stored by each battery item as it ran as the basis for a data management scheme. Experimenters using data stored by battery items were to use Smalltalk-80 itself to query this data base.

The most efficient way to construct an aggregate object in Smalltalk is to define a class with a field (or instance variable) for each element of the aggregate. This method of defining aggregates resembles the record mechanism of Simula-67.

The use of records introduces a number of limitations. For instance, it is impossible to add a field to a Smalltalk record on a per object basis, and it is quite difficult to iterate (**do:**) over all the fields of a record. Adding a field on even a per class basis requires the definition of a new subclass. Motivations for adding fields to a record might include adding a timestamp, or a link to a descriptor giving help and limit information.

Smalltalk dictionaries do not share these difficulties. It is very easy to add a new field to a dictionary. One simply stores into an unnamed slot. Iteration over the elements of a dictionary simple in Smalltalk.

Extension and iteration over objects in the ways described above resemble a coding style frequently seen in Lisp. The property list mechanism in Lisp is often used to allow the addition of arbitrary attributes to individual data structures. The properties of an object can be iterated across as well. Interestingly, earlier versions of Smalltalk (such as Smalltalk-72 [Goldberg 76]) showed a much stronger Lisp influence than do later versions.

Earlier versions of the battery simulation used dictionaries for parameter and data management. Managing parameter and data management objects as dictionaries introduced a number of complications. One was that access to fields required a somewhat more cumbersome syntax. This was a significant problem because it complicated the syntax of user queries to the data management system. More seriously, the process of constructing these dictionaries required explicit code to initialize the fields that seemed to wastefully recapitulate exactly the mechanism that instance variable inheritance provided.

The table below summarizes the trade-offs between using Lisp-style records (instance variables) and P-List style dictionaries in Smalltalk. (The entry for redefinability refers to the ease with which one can interpose a calculation for a simple instance variable reference.

Feature	Instance Vars	Dictionaries
iteration (**do:**)	hard	easy
extension (adding a field)	hard	easy
efficiency (speed/space)	high	low
access syntax	clean	messier
redefinability (access side effects)	high	lower
instantiation overhead	low	higher

The solution I finally opted for uses a new pair of classes, AccessableObject and AccessableDictionary, to allow dictionary-like access to objects, and object-like access to dictionaries

Accessable objects allow dictionary-style access to all their instance variables, along with record-style access to a built in dictionary. Hence, instance variables can be accessed using **at:** and **at:put:**, as well as the standard record-style access protocol (**name** and **name:**).

Both access styles are provided without any need to explicitly define additional accessing methods. The record-style access method is rather slow however, and should be overridden when efficiency is an important consideration.

If **name:** or **at:put:** storage attempt is made and no instance variable with the given name exists, an entry is made for the given selector in the AccessableObject's item dictionary. Thereafter, this soft instance variable may be accessed using <u>either</u> <u>access</u> <u>method</u>. In this way, uniform access to hard and soft fields is provided, and Smalltalk's Simula/Lisp dualism is bridged. AccessableObjects provide a way of adding associations to objects in a manner similar to that provided by Lisp's property list mechanisms.

The example below shows some of the capabilities of AccessableObjects:

AccessableObject class methods for: examples

```
    example
        "AccessableObject example"

        | temp |
        temp ← AccessableObject new.
        temp dog: 'Fido'.
        temp cat: 'Tabby'.
        Transcript print: temp dog; cr.
        Transcript print: temp items; cr.
        temp keysDo: [:key | Transcript print: key; cr].
        Transcript print: (temp variableAt: #items); cr.
        Transcript endEntry
```

I think in hindsight that another reason for the difficulty of parts of the battery simulation discussed above was that I was violating Einstein's edict that: "Things should be as simple as possible, but not simpler." I was attempting to design a single, uniform, all encompassing access mechanism to solve an unwieldy collection of distinct problems. A more natural solution might have involved several different data structures and access mechanisms. The scheme I was trying to construct resembled the multiple view mechanisms seen in some data base systems. It might have been easier in the end to treat the parameter storage requirements of the battery items as distinct from the data management problems in both the battery items themselves and in the data management system. As long as data could be easily converted from one format to another, this sort of approach will work. Such an approach would have led to the explicit entry of battery data into an object-oriented data base, rather than an attempt to extend the original objects so that they themselves became the database.

I think this is an observation that is applicable to a wide range of object-oriented design problems. The power of object-oriented systems can delude one into searching for a single, all encompassing solution to too wide a collection of problems. In the end, a number of freely convertible currencies may prove easier to manage than a single common currency. I could easily take the opposite side of this argument. (Indeed, in the battery simulation described herein, I clearly did.)

Looking for shortcuts like the one described above reminds me of Henry Hudson and the navigators who searched unsuccessfully for the Northwest Passage. Hudson, on a journey to find a Northeast passage around Asia at one point abandoned this search, and set sail across the Arctic to search once more for a Northwest passage (a rather dramatic change in his itinerary). Hudson's efforts to find the Northwest passage finally came to a tragic end in 1611 in the Canadian bay that bears his name. The crew of his ship, the Discovery, mutinied, and left Hudson, his son, and a few loyal crew members adrift in a lifeboat in Hudson Bay.

Hudson was not the last victim claimed by the search for the Northwest Passage. Many more perished before Admunsen finally completed a journey of the passage during the period from 1903 to 1905.

During 1958 and 1959, the American nuclear submarine Nautilus completed a journey from the Atlantic to the Pacific <u>under</u> the Arctic icecap, as part of the celebration of the International Geophysical Year. In 1969, the icebreaker Manhattan "humped" its way across the Northwest Passage.

There must be a lesson here someplace. The success of the Nautilus and the Manhattan probably says something about the wisdom of tackling tough problems with appropriately powerful tools. Until such tools are available, one's prospects for success may be no better than Hudson's.

There appear to be approaches on the horizon that address some of the uniform access issues mentioned above. Delegation-based object-oriented languages may allow some of the sorts of per instance manipulations that are difficult in Smalltalk. (See [Lieberman 86] and [Ungar 87].) Techniques like the use of (database style) views [Garlan 86] and metaclass manipulations like reflection [Maes 87] might have allowed some of the goals of my original, ambitous data management ideas to be realized.

O^2 Programming is Easy. O^2 Design is Hard

I think I might summarize my experiences with object-oriented programming and design as follows: object-oriented programming is easier. Much easier. Object-oriented design is harder. Much harder.

Once one has put in the necessary time to master the class libraries, routine programming tasks seem to take remarkably little time. I found that the aggregate handling capabilities of the Collection hierarchy in particular saved me enormous amounts of time. I found myself making statements like: "I don't write symbol table managers anymore. I'm a Smalltalk programmer. I have the Dictionary classes." In fact, my subjective impression was that programming time itself ceased to be a major determinant of how long it would take me to accomplish a programming task. Instead, the process seemed to become completely design limited.

The reasons for the difficulty of object-oriented design do not reflect poorly on it, though, since they result from the fact that object-oriented techniques allow one to address much more difficult problems than were here-to-fore addressable.

(On those occasions where I've felt at ease taking a short-sighted, slash and burn attitude towards Smalltalk programming, I've found that quick-and-dirty design can be just as easy as quick-and-dirty programming. It is this aspect of Smalltalk that, I would guess, makes it popular for prototyping.)

Certainly a major reason that object-oriented programming is design limited is one that has already been discussed at length here: the tension between simplicity and task specificity on one hand vs. generality and potential future applicability on the other hand. Since polymorphism and inheritance combine to make efforts to make a component more general much more likely to pay off, the designer can do worse than spend time considering what he or she might do achieve such generality.

Another reason that object-oriented design takes time is the great wealth of design alternatives that can be brought to bear on a given problem. In a traditional system, the designer might bemoan the fact that there is simply <u>no</u> good way to resolve a given design difficulty. In an environment like Smalltalk's, there might be several different ways of approaching a given problem, each with its own strengths and weaknesses. The thoughtful designer must attempt to weigh all such alternatives. Selecting from among such an embarrassment of alternatives can be time consuming. Whether there are no solutions or an infinite number of solution to a linear system, the determinant of that system will still be zero.

It is these sorts of design decisions that can make object-oriented design both aggravating and exhilarating. A tune by the new wave band Devo summed this dilemma up nicely with the lines:

What we've got is freedom of choice
What we want is freedom from choice

The need to arbitrarily relax design constraints can haunt the designer when combined with attempts to generalize components to meet hypothetical future requirements. One is often left with the vague feeling that each fork in the design road, however innocuous, might be the one that preempts some major unseen design simplification somewhere down the road. It is for this reason that it is frequently better to allow general components to emerge from experience rather that to attempt to design them in a strictly top-down fashion.

The final reason for the difficulty of object-oriented design is simply that these techniques that allow us to confront complex, design bound problems that have been beyond the reach of conventional programming techniques. For example, the problems of deciding where to trade-off design simplicity and generality in certain components don't arise to the same extent in conventional systems, since the same high reuse potential is simply not there.

It would seem object-oriented techniques offer us an alternative to writing the same disposable programs over and over again. We may instead take the time to craft, hone, and perfect general components, with the knowledge that our programming environment gives us an the ability to reexploit them. If designing such components is a time consuming experience, it is also the sort of experience that is aesthetically satisfying. If my alternatives are to roll the same rock up the same hill every day, or leave a legacy of polished, tested general components as the result of my toil, I know what my choice will be.

Chapter VII -- Conclusion

The challenges associated with application environments characterized by requirements that evolve rapidly are not well addressed by traditional software design and software engineering techniques. The use of object-oriented frameworks fills the large gap that exists between application programs and application independent components, and provides a path along which applications may evolve as they mature. By allowing both application skeletons and components to serve as the roots of frameworks, object-oriented inheritance allows previously application specific skeletons to be generalized, and currently application independent components to be tailored to specific applications. The reuse potential of both is hence greatly increased. A higher reuse potential, in turn justifies a higher level of effort to polish and perfect such components, since any application that uses such a component will benefit from such changes. Inheritance hierarchies provide a way in which one can manage the evolution of a family of related applications as they diverge.

Frameworks promise to have their greatest impact where the need to confront volatile requirements is the greatest: in scientific research programming (such as neural network simulation, for example), in certain realtime embedded application domains, and in experimental user interface prototyping environments, to name but a few.

Frameworks, along with other approaches made possible by object-oriented techniques (such as our Plumbing-Support classes), seem to promise to be of major benefit in realtime application environments, despite potential performance problems. This is because performance is but one of the requirements that distinguishes realtime environments from more prosaic application domains.

The Battery simulation described herein, along with the CPL Battery work that preceded it, have conclusively convinced me that object-oriented techniques and systems can be of major benefit to designers and implementors of realtime scientific laboratory packages. The use of object-oriented frameworks turned what would have been a disjoint collection of individual applications into an integrated system with a high degree of code sharing. This economy, in turn, allowed effort to be expended on "luxuries" like a better user interface, including menus and a help system. It also allowed each application to share more complex data encoding and data management schemes, and a table driven parameter editor. These in turn allowed each application to implement a wider range of experimental designs than would otherwise have been possible. I believe that none of this would have been feasible without the framework-based structure present in both Battery implementations.

My experience with the resilience of these techniques in the face of new design requirements has been very positive The hierarchical Battery frameworks have gracefully accommodated dramatic alterations in the original requirements, such as the addition of digitized speech stimuli and a variety of complex realtime displays. These features, once added, are available to all the experimental designs that might benefit from them.

The original Battery experience is evidence as well that object-oriented techniques can be of enormous utility even when no object-oriented language is available.

Designing to facilitate change requires a great deal of care and foresight. Design must be seen not as a stage that occurs near the start of the software lifecycle, but as a process that pervades it. Object-oriented tools and techniques make it much more likely that an investment in design time and effort will be repaid in terms of greater system generality. There are few feelings more gratifying to the software designer than realizing that through prudence and foresight, one has at hand components that solve nearly all of what would otherwise have been a tedious reimplementation of some boiler-plate piece of code. Object-oriented frameworks and careful design can together help to bring about such a pleasant state of affairs where such an outcome would otherwise not have been possible.

References

[Apple 84]
Apple Computer, Inc.
Lisa Toolkit 3.0
Apple Computer, Cupertino, CA

[Apple 85]
Apple Computer, Inc. (] Rose et. al.)
Inside Macintosh Volumes I, II, and III
Addison-Wesley, Reading, MA, 1985

[Apple 86]
Apple Computer, Inc. (Dan Ingalls et. al.)
Smalltalk-80 for the Macintosh Version 0.3
Apple Programmer's and Developer's Association
Renton, WA, 1986

[Bassett 87]
Paul G. Bassett
Frame-Based Software Engineering
IEEE Software, Volume 4 Number 4, July 1987 pages 9-16

[Beyer 75]
Terry Beyer
FLECS User Manual
U. of Oregon, Eugene, OR, 1975

[Bhasar 86]
K. S. Bhasar, J. K. Pecol) and J. L. Beug
Virtual Instruments: Object-Oriented Program Synthesis
OOPSLA '86 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 303-314

[Boehm 85]
Barry W. Boehm
A Spiral Model of Software Development and Enhancement
TRW tech. report 21-371-85
TRW Inc. 1985

[Borning 79]
Alan Borning
ThingLab -- A Constraint Oriented Simulation Laboratory
Technical Report No. SSL-79-3
Xerox Palo Alto Research Center
July 1979

[Borning 82]
A. H. Borning and D. H. H. Ingalls
A Type Declaration and Inference System for Smalltalk
9th POPL, 1982, 133-141

[Borning and O'Shea 86]
A. Borning and T. O'Shea
DeltaTalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80 Language
unpublished. 1986

[Borning 86]
Alan Borning
Classes versus Prototypes in Object-Oriented Languages
Proceedings of the ACM/IEEE Fall Joint Computer Conference
Dallas, TX, November 1986, 36-40

[Brooks 75]
Frederick P. Brooks
The Mythical Man-Month: Essays on Software Engineering
Addison-Wesley, Reading MA, 1975

[Brooks 87]
Frederick P. Brooks
No Silver Bullet: Essence and Accidents of Software Engineering
IEEE Computer, Vol. 20, No. 4, April 1987

[Brown 84]
Dave Brown
Why We Did So Badly in the Design Phase
Software Engineering Notes, Volume 9, Number 4,
July 1984, pages 58-60

[Chernicoff 85]
Stephen Chernicoff
Macintosh Revealed Volume One: Unlocking the Toolbox
Macintosh Revealed Volume Two: Programming with the Toolbox

Hayden, Hasbrouck Heights, NJ, 1985

[Cook 87]
William R. Cook
A Denonational Semantics of Inheritance (Extended Abstract)
Brown University, 23 July 1987

[Cox 86]
Brad Cox
Object-Oriented Programming: An Evolutionary Approach
Addison-Wesley, 1986

[Deutsch 83]
L. Peter Deutsch
The Dorado Smalltalk-80 Implementation:
Hardware Architecture's Impact on Software Architecture
In Glenn Krasner, *Smalltalk-80: Bits of History, Words of Advice*
Addison-Wesley, Reading, MA, 1983

[Donchin 75]
Emanuel Donchin and Earle F. Heffley
Minicomputers in the Signal Averaging Laboratory
American Psychol 30: 299-312, 1975

[Donchin 81]
Emanuel Donchin
Surprise! ... Surprise?
Psychophysiology 19: 493-513, 1981

[DeRemer 76]
Frank DeRemer and Hans H. Kron
Programming-in-the-Large Versus Programming-in-the-Small
In AFIPS Conference Proceedings, pages 425-430, AFIPS, 1975.
NCC 1975, Anaheim, CA

[Duncan-Johnson 77]
Connie Duncan-Johnson and Emanuel Donchin
On Quantifying Surprise:
The variation in event-related potentials with subjective probability
Psychophysiology: 14: 456-467, 1977

[Fairley 85]
Richard E. Fairley
Software Engineering Concepts

McGraw-Hill, New York, NY, 1985

[Foote 85]
Pearl II Subroutine Library
CPL Technical Report
Cognitive Psychophysiology Laboratory, Dept. of Psychology
University of Illinois at Urbana-Champaign
November 1985

[Foote 87]
Brian Foote
A Smalltalk-80 Protocol Browser
CS327 (Software Engineering) Class Report
Dept. of Computer Science, University of Illinois at Urbana-Champaign
17 May 1987

[Garlan 86]
David Garlan (CS, CMU)
Views for Tools in Integrated Environments
IFIP WG2.4 International Workshop on Advanced Programming Environments
Trondheim, Norway, June 1986

[Goldberg 76]
Adele Goldberg and Alan Kay, editors
with the Learning Research Group
Smalltalk-72 Instruction Manual
Xerox Palo Alto Research Center

[Goldberg 83]
Adele Goldberg and David Robson
Smalltalk-80: The Language and its Implementation
Addison-Wesley, Reading, MA, 1983

[Goldberg 84]
Adele Goldberg
Smalltalk-80: The Interactive Programming Environment
Addison-Wesley, Reading, MA, 1984

[Halbert 87]
Daniel C. Halbert and Patrick D. O'Brien
Using Types and Inheritance in Object-Oriented Programs
IEEE Software, to appear, 1987

[Harrison 86]
William Harrison (CS Dept. IBM T. J. Watson Research Laboratory)
A Program Development Environment
for Programming by Refinement and Reuse
Proceedings of the Nineteenth Annual Hawaii International
Conference on System Sciences, 1986

[Heffley 85]
Earle F. Heffley, Brian Foote, Tony Mui and Emanuel Donchin
Pearl II:
Portable Laboratory Computer System for Psychophysiological
Assessment using Event Related Brain Potentials
Neurobehavioral Toxicology and Teratology,
Volume 7, pp. 399-407, 1985

[Hooper 86]
Kristina Hooper
Architectural Design: An Analogy in User Centered System Design
in *User Centered System Design*
Edited by Donald A. Normal and Stephen W. Draper
Lawrence Erlbaum Associates, Hillsdale, NJ, 1986

[Ingalls 78]
Daniel H. H. Ingalls
The Smalltalk-76 Programming System Design and Implementation
5th ACM Symposium on POPL, pp. 9-15
Tucson, AZ, USA, January 1978

[Jacobson 86]
Ivar Jacobsen
(Ericsson Telecom and Swedish Institute for Computer Science)
Language Support for Changable Realtime Systems
OOPSLA '86 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 377-384

[Johnson 86a]
Ralph E. Johnson and Simon Kaplan
Towards Reusable Software Designs and Implementations
Proceedings of the Workshop on
Future Directions in Computer Architecture and Software
May 5-7, 1986, Seabrook Island, Charlston, SC
Sponsored by the Army Research Office (ARO)

[Johnson 86b]
Ralph E. Johnson
Type-Checking Smalltalk
OOPSLA '86 Proceedings
Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 315-321

[Johnson 88]
Ralph E. Johnson
Design for Reuse (draft)
University of Illinois, Dept. of Computer Science

[Kaiser 87]
Gail E. Kaiser and David Garlan
Melding Software Systems from Reusable Building Blocks
IEEE Software, Volume 4 Number 4, July 1987 pages 17-24

[Kay 79]
Alan C. Kay
Programming Your Own Computer
The World Book Science Annual, 1979
World Book/Childcraft International Inc., Chicago, IL

[Kerhighan 76]
Brian W. Kernighan and P. J. Plauger
Software Tools
Addison-Wesley, 1976

[Knuth 84]
Donald E. Knuth
Literate Programming
Computing Journal 27, 2 (1984), 97-111

[Kransner 83]
Glenn Krasner
Smalltalk-80: Bits of History, Words of Advice
Addison-Wesley, Reading, MA, 1983

[Lehman 76]
M. M. Lehman and L. A. Belady
A model of large program development
IBM System J., 15(3), 225-52, 1976

Lehman 80]
M. M. Lehman
Programs, life cycles and the laws of software evolution
Proc. IEEE 68(9), 1060-76, 1980

[Lieberman 86b]
Henry Lieberman
Using Prototypical Objects to Implement Shared Behavior
in Object-Oriented Systems
OOPSLA '86 Proceedings
Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 2-4-223

[Lientz 80]
B. Lientz and E. Swanson
Software Maintenance Management
Addison-Wesley, Reading, MA, 1980

[Lubars 86]
Mitchell D. Lubars
A Knowledge-based Design Aid for the Construction of Software Systems
Ph. D. Thesis, Dept. of Computer Science
University of Illinois at Urbana-Champaign
UIUCDCS-R-86-1304

[Maes 87]
Pattie Maes (Vrije Universiteit Brussel)
Concepts and Experiments in Computational Reflection
OOPSLA '87 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Orlando, FL, October 4-8 1977 pages 147-155

[McCracken 82]
Daniel D. McCracken and Michael A. Jackson
Life Cycle Concept Considered Harmful
Software Engineering Notes, Volume 7, Number 2,
April 1982, pages 29-32

[Mittal 86]
Sanja Mittal, Daniel G. Bobrow, and Kenneth M. Kahn
Virtual Copies: At the Boundary Between Classes and Instances
OOPSLA '86 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 159-166

[Nestor 86]
John R. Nestor (Software Engineering Institute, Carnigie-Mellon U.)
Toward a Persistent Object Base
1986

[O'Shea 86]
Chair: Tim O'Shea, with Kent Beck, Dan Halbert and Kurt J. Scmucker
Panel on: The Learnability of Object-Oriented Programming Systems
OOPSLA '86 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 502-504

[Parikh 84]
Girish Parikh
What is Sofware Maintenance Really? What is in a Name?
Software Engineering Notes, Volume 9, Number 2,
April 1984, pages 114-116

[Parikh 85]
Girish Parikh
SOFTWARE MAINTENANCE NEWS
Software Engineering Notes, Volume 10, Number 2,
April 1985, pages 58-60

[Pascoe 86]
Geoffrey A. Pascoe (Productivity Products International)
Encapsulators: A New Software Paradigm in Smalltalk-80
OOPSLA '86 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Portland, OR, September 29-October 2 1986, pages 341-346

[Pressman 82]
Roger S. Pressman
Software Engineering: A Practitioner's Approach
McGraw-Hill, New York, NY, 1982

[Riddle 84]
William E. Riddle
The Magic Number Eighteen Plus or Minus Three:
A Study of Software Technology Maturation
Software Engineering Notes, Volume 9, Number 2,
April 1984, pages 21-37

[Ross 85]
Douglas T. Ross (SofTech, Inc.)
Applications and Extensions of SADT
IEEE Computer, April 1985

[Schmucker 86]
Kurt J. Schmucker
Object-Oriented Programming for the Macintosh
Hayden, Hasbrouck Heights, NJ, 1986

[Smith 87]
Randall B. Smith
Experiences with the Alternate Reality Kit:
An Example of the Tension Between Literalism and Magic.
CHI+GI 1987 Conference Proceedings

[Sommerville 85]
Ian Sommerville
Software Engineering (Second Edition)
Addison-Wesley, Reading, MA, 1985

[Stambaugh 87]
Thomas M. Stambaugh
A Distributed Application Environment: An Architecture
Institute for Research in Information and Scholarship (IRIS)
Brown University, 1987

[Stefik 86a]
Mark Stefik and Daniel G. Bobrow
Object-Oriented Programming: Themes and Variations
AI Magazine 6(4): 40-62, Winter, 1986

[Stefik 86b]
M. Stefik, D. Bobrow and K. Kahn
Integrating Access-Oriented Programming into
a Multiprogramming Environment
IEEE Software, 3, 1 (January 1986), 10-18

[Sternberg 69]
Saul Sternberg
Memory Scanning:
mental processes revealed by reaction time experiments
Am Scientist 57: 421-457, 1969

[Stroustrup 86]
Bjarne Stroustrup
The C++ Programming Language
Addison-Wesley, Reading, MA, 1986

[Tracz 87]
Will Tracz
Resuability Comes of Age
IEEE Software, Volume 4 Number 4, July 1987 pages 6-8

[Ungar 87]
David Ungar and Randall B. Smith
Self: The Power of Simplicity
OOPSLA '87 Proceedings
(Object-Oriented Programming Systems, Languages and Applications)
Orlando, FL, October 4-8 1977 pages 227-242

[Zdonik]
Stanley B. Zdonik (Brown U.)
Version Management in an Object-Oriented Database