1991

Object-oriented menu-driven front-end for simulation of manufacturing systems

Pamela S. Woodbury
Lehigh University

OBJECT-ORIENTED MENU-DRIVEN FRONT-END FOR SIMULATION OF MANUFACTURING SYSTEMS

by

Pamela S. Woodbury

A thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Electrical Engineering

Lehigh University

(October, 1990)

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Masters of Science in Electrical Engineering.

10/29/90

Date

Edwin J. Kay

Advisor in Charge

Lawrence J. Varnerin

CSEE Department Chairperson

ACKNOWLEDGEMENTS

Their are many people that I would like to thank for their support while completing my thesis. First of all is my extremely supportive husband and my family. With out their encouragement along the way, this project might not have ever gotten off the ground.

I would also like to thank AT&T Bell Laboratories for their financial and emotional support. I wish to express thanks to both my supervisor and my co-workers for all of their uplifting encouragement.

Finally, I would like to thank my advisor, Professor Edwin Kay, for his guidance in my work. Thanks to his advice and guidance, this project was a success.

Table of Contents

Abstract

Computer simulation can be used to asses the performance of complex production systems and to identify their design flaws and operating problems. The problem is that simulation systems are typically very complex and not very user friendly.

This thesis presents a high level design for an object-oriented, menu driven front-end of a simulation package for one type of production system, the automated flow line. The front-end is designed using object classes provided for simulation in the Smalltalk-80 environment. The focus of the design is on providing a user friendly means of entering machine and buffer storage data into the simulation. This was accomplished by the use of menus. The user first selects an action from the main menu and is then guided through that selection by the use of menu choices or prompts for specific information.

I. Introduction

In today's factories there are many types of manufacturing systems. There are high volume production systems which are used when there is high demand for a product and correspondingly little variation in production. At the opposite extreme there are flexible manufacturing systems that must produce a variety of products with virtually no time lost for change over from one product to the next. Computer simulation can be used to assess the performance of these complex production systems and to identify their design flaws and operating problems. [Groover, 1987]

One type of mass-production system, known as an automated flow line, consists of several machines that are linked together by work-handling devices that transfer parts between the machines. The parts are transfered automatically and the machines carry out their tasks automatically. A raw part enters the line at one end and the processing steps are performed sequentially as the part moves from one station to the next. It is possible to have buffer storage zones between the machines where parts are stored before being processed by the next station. It is also possible to include inspection stations and manual work stations in the flow line.[Groover, 1987]

In an automated flow line, it is important to balance the flow in the line so that all process steps are of approximately equal capacity.

because any additional capacity beyond that of the least productive step is usually wasted. Computer simulation can aid the system designer by providing specific information regarding where machines may need to be added to improve throughput. Simulation can also provide information about the line such as: reliability, line performance, and how much improvement might be made by providing storage buffers.

Simulation systems are typically very complex and not very user friendly. Because of time constraints present in business today, an easier and faster method of modeling systems is needed. [Grant, 1988]

A high level design for an object-oriented, menu driven front-end, of a simulation package for an automated flow line, will be presented in this paper. The front-end will be designed using the object classes provided for simulation in the Smalltalk-80 environment [Goldberg & Robson, 1989]. The design will focus on providing a user-friendly means of entering machines and buffer storage into the simulation. It will not focus on issues such as simulation suspension or simulation output since there is a package called SimTalk[1] that provides support in these areas.

---

[1] SimTalk is a simulation package, available from Tektronix, which runs within the Smalltalk-80 environment. SimTalk defines a large number of predefined objects for use in designing complex simulations. This provides the simulation with inherited capabilities that include graph and histogram displays, animation controllers, random number generators, and probability distributions. SimTalk lets the user suspend simulation, modify the definitions of entities within it, then restart the simulation from the point at which it was suspended. In order to save the results of a simulation, SimTalk

The design is based on Smalltalk-80 because of it's extensibility. In the paper entitled "Object Oriented Simulation of Manufacturing Systems: A Smalltalk Experience", Lawrence Doe [1989] states that "The tools used by the [manufacturing system] designer must provide for ever increasing improvements in productivity". A package designed using Smalltalk-80 would provide this capability, since one of the underlying aspects of Smalltalk-80 is it's ability to add to it's current environment. Individual pieces of the simulation can be implemented and executed without having other pieces of the simulation available. Also, a complete simulation package can easily be enhanced by adding new objects or methods to represent improvements in a manufacturing system.

---

can write its statistical output to a file. The user may also choose to display the statistics in one of six kinds of graphs. SimTalk also provides a way of displaying simulation output in animated form.

II. Smalltalk-80 Overview

The following discussion on Smalltalk-80 is drawn from the book entitled "SMALLTALK-80: The Language". [Goldberg & Robson, 1989]

Smalltalk-80 is an environment in which all components are referred to as objects. Objects are made up of private memory (instance variables) and a set of operations (methods). Objects respond to messages. A message is a request, sent by an object, for another object to execute one of its methods. An important distinction between a message and a method is that a message specifies which operation is desired, while a method is the actual implementation of the operation. It is possible for two different objects to respond to the same message in entirely different ways.

Objects are organized into classes which in turn are organized in a hierarchy so that objects in a subclass will have all of the properties of objects in the superclass. An occurrence of an object described by a class is referred to as an instance of that class. The class is the description of the instances' instance variables, and methods. All instances of the same class, therefore, use the same set of methods to describe their operations.

The subclass hierarchy allows classes to share similar descriptions. The instances of a subclass are the same as the instances of its superclass except for explicitly stated differences. In other words, the subclass inherits both the variables and methods of its superclass. The subclass may declare new variables and it may also add new methods or override existing methods in the superclass. A subclass overrides a superclass method by adding a new method in the subclass with the same message pattern as the method of the superclass. Instances of the subclass will respond to the message by executing the new method of the subclass rather than that of the superclass.

The following is a summary of the Smalltalk-80 terminology as given by Goldberg & Robson.

Summary of Terminology	
object	A component of the Smalltalk-80 system represented by some private memory and a set of operations.
message	A request for an object to carry out one of its operations.
class	A description of a group of similar objects.
instance	One of the objects described by a class.
instance variable	A part of an object's private memory.
method	A description of how to perform one of an object's operations.
subclass	A class that inherits variables and methods from an existing class.
superclass	The class from which variables and methods are inherited.

This paper will present classes in the same manner as does [Goldberg & Robson, 1989]. That is, a class will be presented in two forms. One form will describe the functionality of the instances and the other will describe the implementation of that functionality.

The *protocol description* lists the messages that an instance of that class can respond to. Each message has a description of the operation to be performed when the message is received. An *implementation description* shows the implementation of the functionality described in the protocol description.

In some cases only the protocol description of the methods will be provided, as this description will be adequate to present the context of the information. In other cases, where more detail is required, both the protocol description and the implementation description will be provided. Italics are used in the implementation descriptions to indicate pseudo code where actual methods have not been defined.

In Smalltalk-80 there are three types of messages: unary, binary, and keyword. A unary message is a message that has no arguments, a binary message takes one argument and is composed of either 1 or 2 special symbols, and a keyword message takes as many arguments as there are keys. Some examples of each are shown in table 1.

mesage type	message	message expression	response
unary	sqrt	4 sqrt	2
binary	+	4 + 5	9
keyword	quo:	6 quo: 2	3
keyword	newDay: month: year:	Date newDay: 6 month:#Feb year: 91	6-Feb-91

Table 1
Message Examples

To develop an application in Smalltalk-80, the programmer modifies the existing environment. To achieve the desired results, new objects and classes, with corresponding variables and methods, may be added to the system. Subclasses of existing classes may be added, which enhance or override the existing superclass description, or existing classes and methods may simply be rewritten. This last method of changing the environment is not a highly recommended one, since it may be possible that the particular method being changed is being used in more than one application. To ensure the integrity of the existing system, it is wiser to create a subclass and override that method in the subclass.

The factory simulation front-end developed in this paper is done by creating subclasses of existing classes. It is therefore necessary to first define some of the tools for simulation in the Smalltalk-80 environment. After that, the factory line for which this front-end is designed will be described, followed by a high-level description of that front-end. The protocol descriptions of the classes added to the system will then be presented along with some of the corresponding implementation descriptions.

Before proceeding, a brief description of the Smalltalk-80 environment is presented. The Smalltalk-80 environment has a highly interactive user interface, with pull-down menus and multiple windows. Smalltalk-80 is usually run on a system with a 3 button mouse. Smalltalk refers to these buttons as the red, yellow and blue buttons. See figure 1 for the layout of the buttons on the mouse.

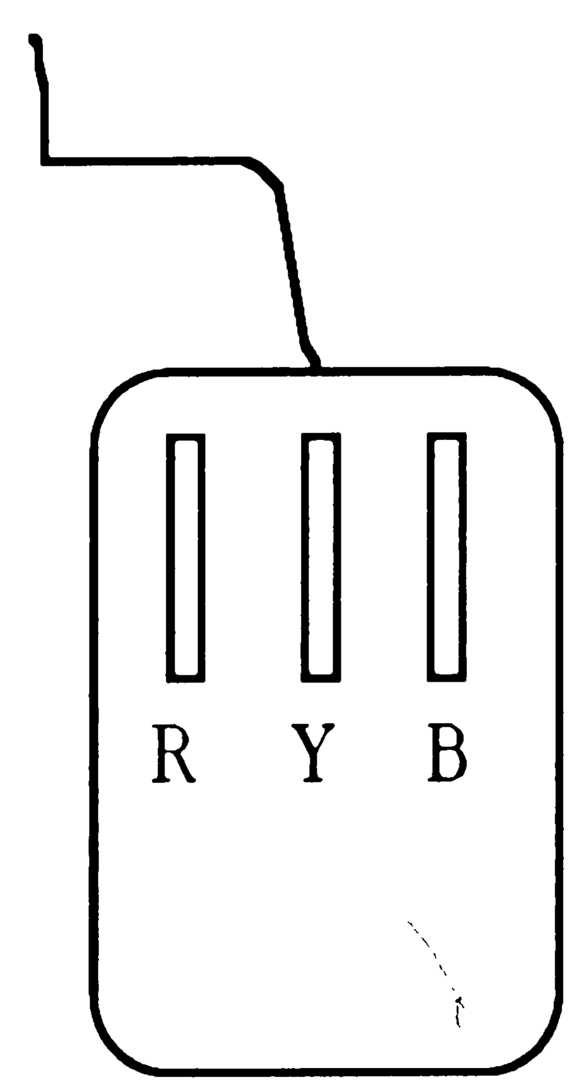


Figure 1
Mouse Buttons

By convention, each mouse button has a unique function in the system. The blue button is used for window control: closing, resizing, moving, etc.. The red button controls the text cursor when the mouse is in a text window. The yellow button controls functions within the window. The options for this button differ according to the purpose of the window. For example, in a text window, the yellow button controls the functions such as copying, deleting, pasting, etc.. The actual functionality of the mouse buttons is decided upon by the software designer. However, the assignment of functions described above is done for the sake of consistency.

III. Smalltalk-80 Simulation Framework

Smalltalk-80 provides a framework for event-driven simulations. Goldberg & Robson describe event-driven simulations as "simulations in which a collection of independent objects exist, each with a set of tasks to do, and each needing to coordinate its activity's times with other objects in the simulated situation". The two major classes provided for simulation in Smalltalk-80 are class **Simulation** and class **SimulationObject**. The class **SimulationObject** describes the kind of object to appear in the simulation. These objects have a set of tasks to carry out, and the methods for these objects describe the manner in which the tasks are completed. An instance of class **Simulation** is the driver of the simulation. It is responsible for maintaining the simulated clock, the queue of events, and the coordination of arrival of new objects and resources into the system.

Class **Simulation** and class **SimulationObject** are abstract classes because they are not intended to have instances. According to [Goldberg & Robson, 1989] an abstract class "provides a framework for a method that is refined or actually implemented by the subclass". If an instance of one of these classes was created, it would not be able to respond to all of the messages successfully because some of the messages are not implemented at this level. It is therefore the responsibility of the subclass to implement these messages.

Following is a partial protocol description of class **SimulationObject** and class **Simulation**. Only the messages that are necessary to understand the design of the front-end to be described later are presented. For a complete description of these classes refer to [Goldberg & Robson, 1989]. The following descriptions are presented here exactly as they are found in the Goldberg & Robson text.

SimulationObject instance protocol

initialization	
initialize	Initialize instance variables, if any.
simulation control	
startUp	Initialize instance variables. Inform the simulation that the receiver is entering it, and then initiate the receiver's tasks.
tasks	Define the sequence of activities that the receiver must carry out.
finishUp	The receiver's tasks are completed. Inform the simulation
task language	
holdFor: aTimeDelay	Delay carrying out the receiver's next task until aTimeDelay amount of simulated time has passed.

acquire: amount ofResource: resourceName

Ask the simulation to provide a simple resource that is referred to by the String, resourceName. If one exists, ask it to give the receiver amount of resources. If one does not exist, notify the simulation user (programmer) that an error has occured.

produce: amount ofResource: resourceName

Ask the simulation to provide a simple resource that is referred to by the String, resourceName. If one exists, add to it amount more of its resources. If one does not exist, create it.

inquireFor: amount ofResource: resourceName

Answer whether or not the simulation has at least a quantity, amount, of a resource referred to by the String, resourceName.

Simulation instance protocol

initialization

initialize	Initialize the receiver's instance variables.

modeler's initialization language

defineArrivalSchedule	Schedule simulation objects to enter the simulation at specified time intervals, typically based on probability distribution functions. This method is implemented by subclasses.
defineResources	Specify the resources that are initially entered into the simulation. These typically act as resources to be acquired. This method is implemented by subclasses.

modeler's task language

produce: amount of: resourceName

An additional quantity of amount of a resource referred to by a String, resourceName, is to be part of the receiver. If the resource does not as yet exist in the receiver, add it; if it already exists, increase its available quantity.

scheduleArrivalOf: aSimulationObject at: timeInteger

Schedule the simulation object, aSimulationObject, to enter the simulation at a specified time, timeInteger.

accessing

time

Answer the receiver's current time.

simulation control

startUp

Specify the initial simulation objects and the arrival schedule of new objects.

proceed

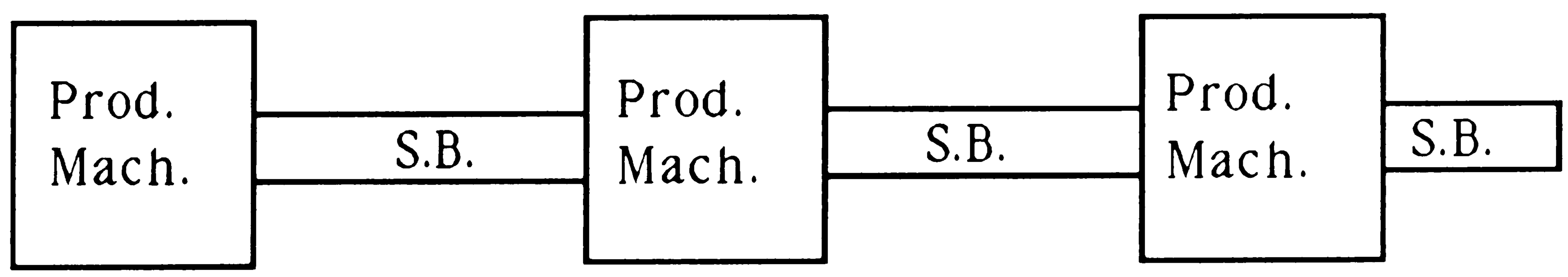
This is a single event execution. The first event in the queue, if any, is removed, time is updated to the time of the event, and the event is initiated.

finishUp

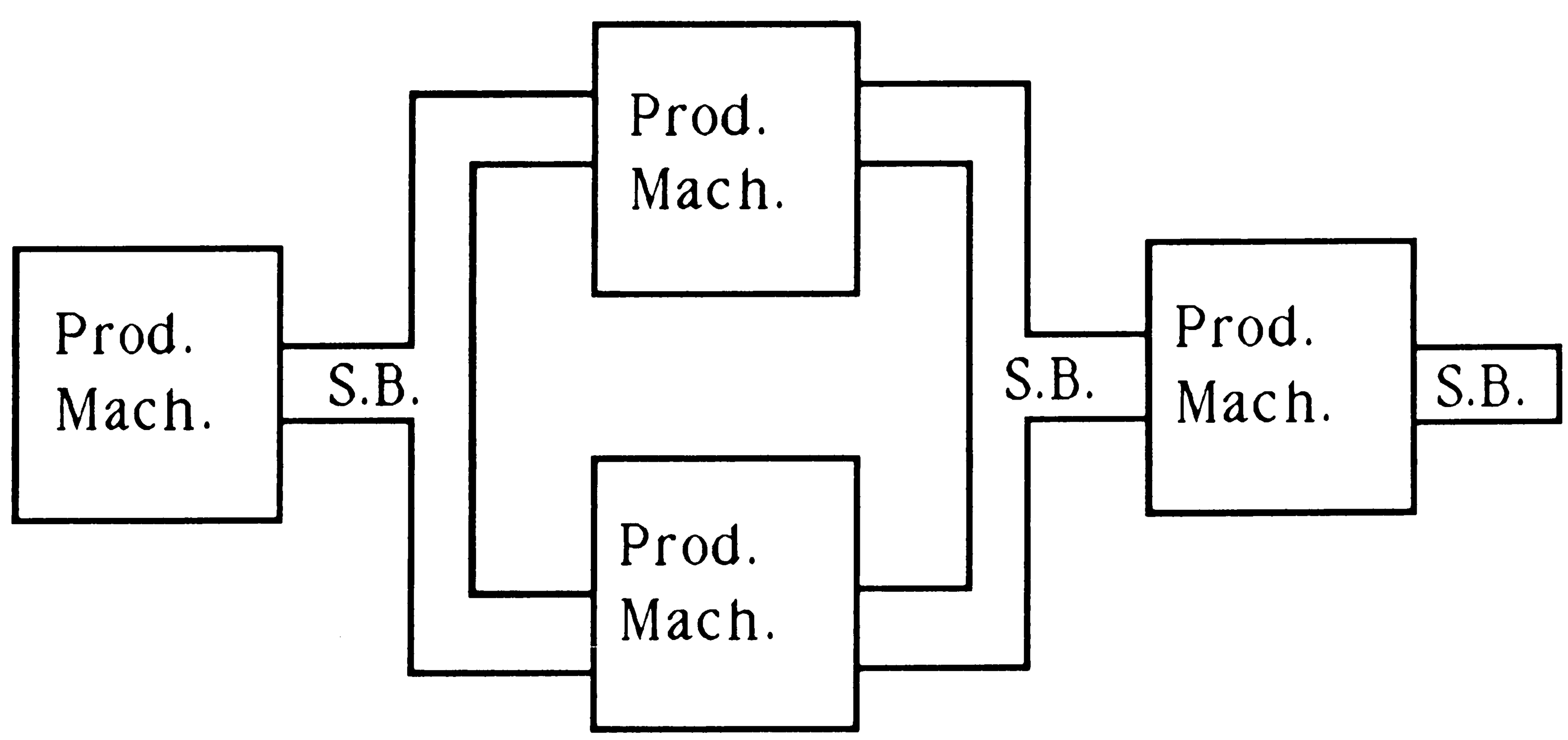
Release references to any remaining simulation objects.

IV. Scope of Manufacturing System to be Modeled

An automated flow line can be modeled by a series of machines that obtain parts from a storage buffer, hold them for a period of time and then place them into another storage buffer. For the purpose of this paper, it is assumed that each machine will have only one buffer from which it receives parts, and only one buffer to feed parts to. Some examples of flow lines are shown below.



Example 1



Example 2

Each machine has a set of characteristics associated with it. These are: machine cycle time and standard deviation, mean time between failures (mtbf), mean time to repair (mttr), where to obtain incoming parts, and where to place outgoing parts.

The user interface must therefore have the capability of adding machines to the simulation, modifying their parameters, and then running the simulation. A description of the user interface is presented next, followed by a high level description of the Smalltalk implementation of that interface.

V. Front-End Description

The user of the front end is first presented with an empty working window, and a pull down menu that would allow two choices: ***add machine*** or ***quit***. After the first machine has been added, the menu will consist of the options shown in figure 2.

add machine
move machine
move buffer
modify machine
remove machine
run simulation
quit

Figure 2
Main Menu Options

Description of Menu Options

add machine	The characteristics for the machine are entered, and the machine is added to the simulation. The machine and its corresponding buffers are positioned in the display.
move machine	Change the position of the machine in the display.

move buffer	Change the position of the buffer in the display.
modify machine	Change any one of the machines characteristics.
remove machine	Remove a machine from the simulation.
run simulation	Execute the simulation for the length of time indicated by the user.
quit	Remove all machines and buffers. Close simulation window.

During execution of the simulation the user can watch how the parts flow through the system. Each buffer keeps a constant display of how many parts are in it at that time, and the machines change color to indicate when they are out of service. It is important to note that the simulation will run according to the specifications entered when the machine is input. The actual placement of the machines and buffers on the screen has no affect on the simulation flow. This graphical representation is solely for the user's ease in following the flow of parts through the system.

When the simulation completes execution, the display is left with the storage buffers displaying the number of parts left in them at that time. At this point, the user has all of the menu options shown in figure 2. This allows the user to run a simulation, wait for its completion, make modifications, and then re-run the simulation.

VI. Front-End Implementation

As previously mentioned, the front-end will be developed by adding new objects and classes to the existing framework for simulation in Smalltalk-80. The design will be based on the model-view-controller (MVC) paradigm [Pinson, 1987]. The model is the actual simulation. It must maintain data on all of the machines and buffers in the simulation. The view is the graphical representation of the simulation. It displays the machines and buffers where the model informs it to. The controller is the user interface. It controls the flow of information between the model, the view, and the user. The controller controls all of the interaction with the user, however, it basically just feeds all of the information on to the model.

The relationships between the model, the view, and the controller are shown in figure 3. According to Pinson's definition of the view, it is the view's primary responsibility to keep the MVC triad together as a family. "Private data within the view protocol are used to connect the view with its model and controller. In his definition of the controller, he states that "Private data within the controller protocol are used to connect the controller with its model and view". These definitions describe the solid lines of communication shown in figure 3. The dotted line is used to represent the relationship from the model to the view. The model protocol does not have an internal reference to its controller or its view, however, the view needs to be informed when the model changes. To establish this link between the model and the view, a dependency relationship is utilized. This type of dependency is described in class **Object**, which is the superclass of all other classes in the Smalltalk-80 system. When the MVC triad is established, the view sends a message to the model to add the view as a dependent of the model. This action forms the dotted line shown in the diagram. When the model has changed, and the view should be informed, the message **changed** is sent to the model. The response to this message is to send the message **update:** to all of its dependents. It is the responsibility of the view to reimplement the message update: so that upon receipt of this message, the view can redisplay the simulation.

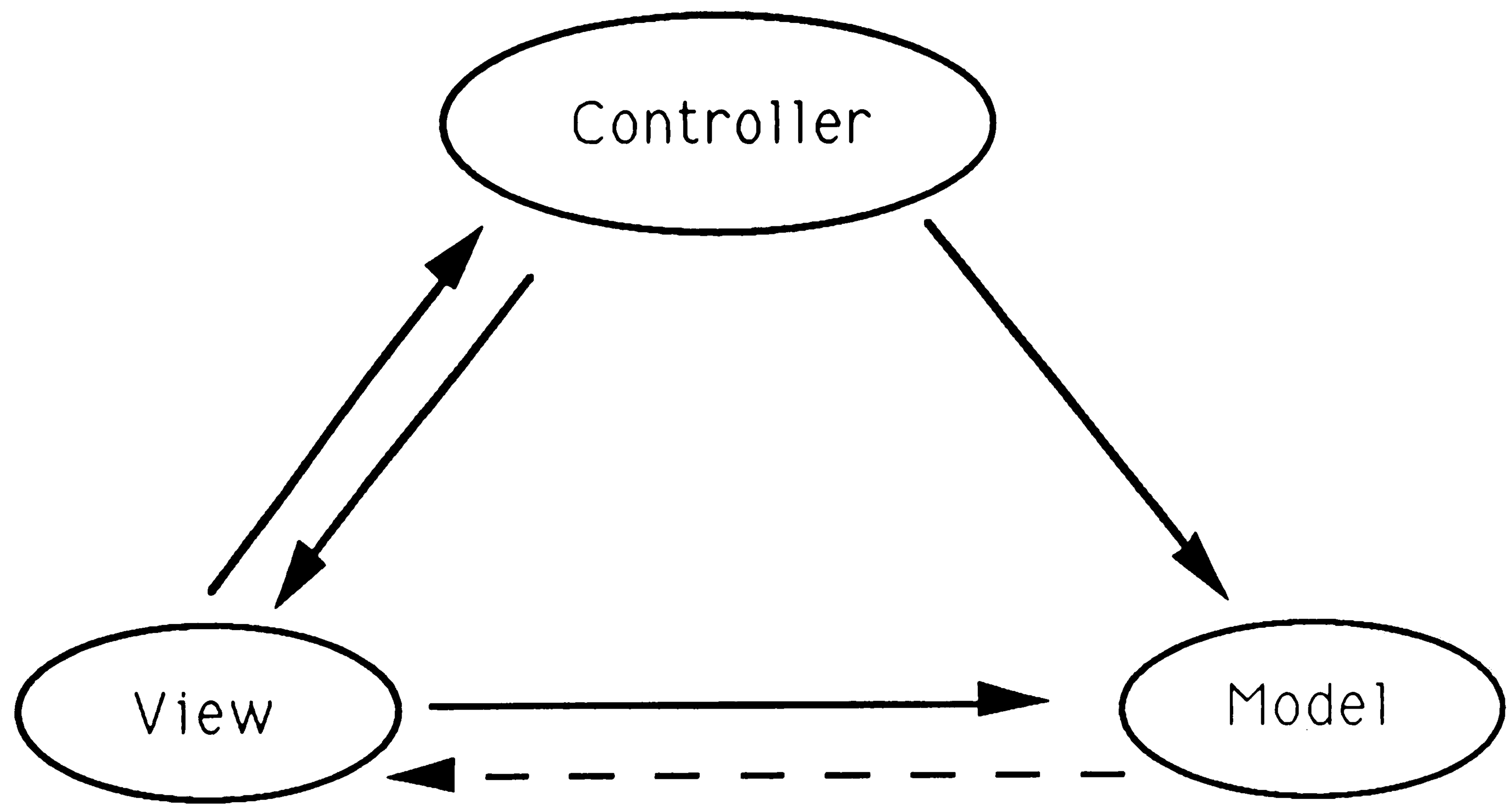


Figure 3
Relationship between the Model,
the View, and the Controller

Figure 4 shows the relationships between the classes and subclasses used in the design of the specific MVC paradigm for this application. This figure should be used as a reference while reading through the class descriptions that follow. The figure shows that the controller, **FactorySimulationController**, is a subclass of the **StandardSystemController** while the view, **FactorySimulationView**, is a subclass of the **StandardSystemView**. The model, **FactorySimulation**, is a subclass of class **Object**. An instance of class **FactorySimulation** has *machines* as an instance variable which is an **OrderedCollection** of instances of **MachineTypeA** or **MachineTypeB**. The other class which is required for this factory simulation application is class **Factory**. Class **Factory**, which is a subclass of class **Simulation**, is responsible for the initialization and maintenance of the actual simulation. Class **Factory** is not shown in Figure 4 since it is the responsibility of the model to send a message to an instance of class **Factory** to start the simulation.

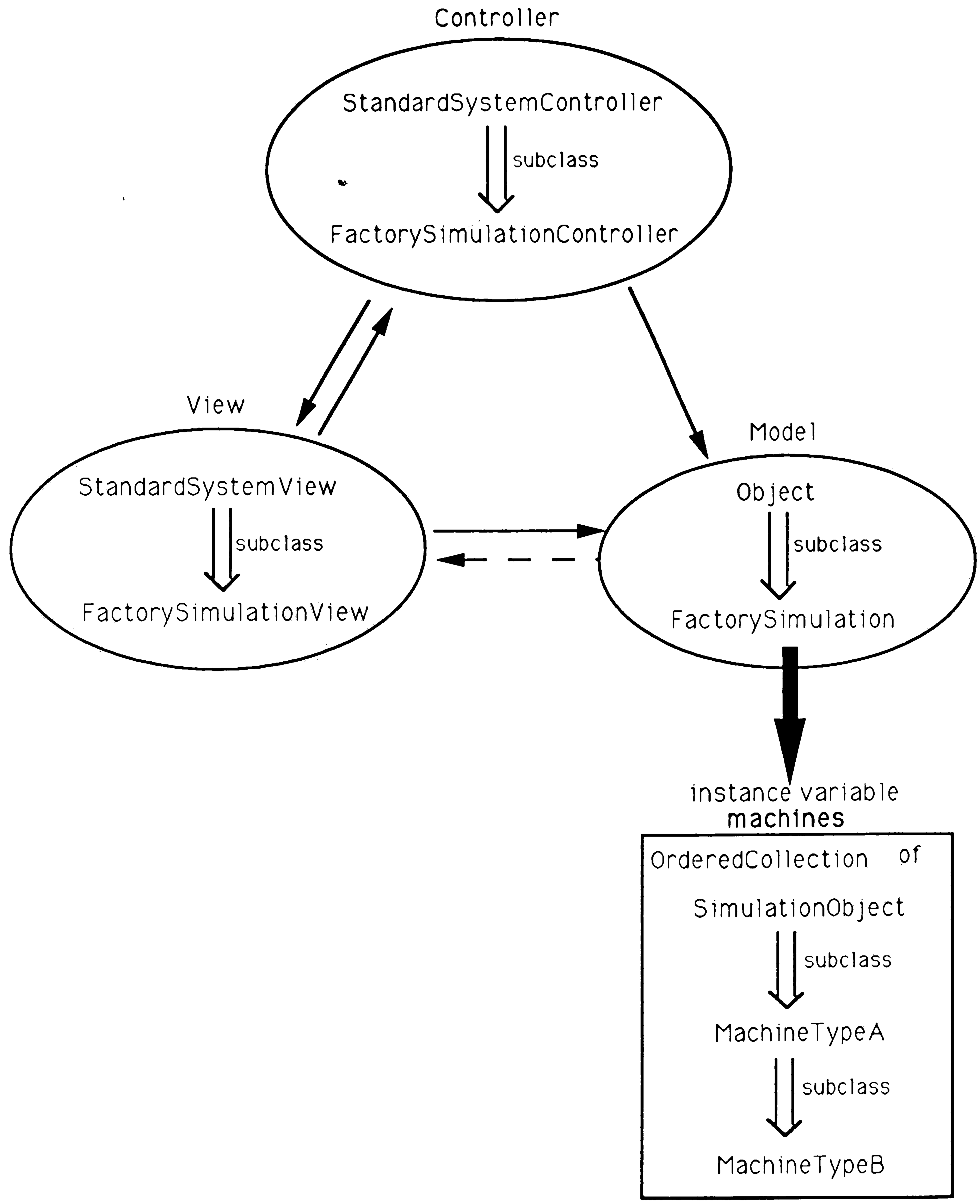


Figure 4
MVC Class/Subclass Relationships

VI.I. The Model

The model needs to represent the machines and the buffers with data structures. As well as having the ability to grow in size, these data structures must be able to remove elements from anywhere in the data structure. For these reasons, an ordered collections [Goldberg & Robson, 1989] are chosen to represent the machines and buffers. These are instance variables *machines* and *buffers* in class **FactorySimulation**

To describe the machine objects, two new classes are added to the system. MachineTypeA is used to represent the first machine or machines in the automated flow line because it does not use an input buffer. MachineTypeB is the more commonly used type in that it uses both input and output buffers. These subclasses have accessing methods for retrieving and modifying the machine parameters. The protocol descriptions of MachineTypeA and MachineTypeB are given next. MachineTypeA is a subclass of SimulationObject. MachineTypeB is a subclass of MachineTypeA because it has all of MachineTypeA's characteristics with the addition of an extra instance variable. By having MachineTypeB be a subclass of MachineTypeA, the repetition of a lot of code is avoided. Note: the instance variables are specified in the implementation descriptions which follow the protocol descriptions for the classes.

MachineTypeA instance protocol

accessing

meanTBF	Return the value of instance variable meanTBF
meanTBF: aNum	Set instance variable meanTBF to the value of aNum
cycleTimeMean	Return the value of instance variable cycleTimeMean
cycleTimeMean: aNum	Set instance variable cycleTimeMean to the value of aNum
cycleTimeDev	Return the value of instance variable cycleTimeDev
cycleTimeDev: aNum	Set instance variable cycleTimeDev to the value of aNum
meanTtoRep	Return the value of instance variable meanTtoRep
meanTtoRep: aNum	Set instance variable meanTtoRep to the value of aNum
outBuf	Return a string with the value of instance variable outBuf
outBuf: aString	Set the instance variable outbuf to the value of aString
machineName	Return a string with the value of instance variable machineName
machineName: aString	Set the instance variable machineName to the value of aString
location	Return the value of instance variable location

location: aPoint	Set the instance variable location to the value of aPoint

simulation control

tasks	The sequence of activities that the machine object must carry out. ie. Determine if the machine is in a failure mode. If yes, hold for a period of time determined by the variable meanTtoRep. If no, hold for an amount of time determined by cycleTimeMean and cycleTimeDev, then add 1 to the quantity in the output buffer.

MachineTypeB instance protocol

accessing

inBuf	Return a string with the value of instance variable inBuf
inBuf: aString	Set the instance variable inbuf to the value of aString

simulation control

tasks	The sequence of activities that the machine object must carry out. ie. Determine if the machine is in a failure mode. If yes, hold for a period of time determined by the variable meanTtoRep. If no, acquire a quantity of 1 from the input buffer, hold for an amount of time determined by cycleTimeMean and cycleTimeDev, then add 1 to the quantity in the output buffer.

Shown next is the implementation description of these two classes.

class name	MachineTypeA
superclass	SimulationObject
instance variable names	meanTBF cycleTimeMean cycleTimeDev meanTtoRep outBuf machineName location

instance methods

accessing

```
meanTBF
	^ meanTBF
meanTBF: aNum
	meanTBF <- aNum
cycleTimeMean
	^ cycleTimeMean
cycleTimeMean: aNum
	cycleTimeMean <- aNum
cycleTimeDev
	^ cycleTimeDev
cycleTimeDev: aNum
	cycleTimeDev <- aNum
meanTtoRep
	^ meanTtoRep
meanTtoRep: aNum
	meanTtoRep <- aNum
outBuf
	^ outBuf
outBuf: aString
	outBuf <- aString
machineName
	^ machineName
machineName: aString
	machineName <- aString
location
	^ location
location: aPoint
	location <- aPoint
```

simulation control

tasks

```
| exponVarTime |
exponVarTime <-
        (Exponential mean: meanTBF) next
                + Simulation active time.
"Must get the time of the simulation from the model"
[Simulation active time >
            dependent simTime]
    whileFalse:
    [[Simulation active time > exponVarTime]
        whileFalse:
        [self holdFor:
            (Normal mean: cycleTimeMean
            deviation: cycleTimeDev) next.
        self produce: 1 ofResource: outBuf.
        "redisplay buffer at this point"
        self changed]
    self holdFor:
        (Exponential mean: meanTtoRep) next.
    exponVarTime <-
            (Exponential mean: meanTBF) next
                + Simulation active time]
```

class name	MachineTypeB
superclass	MachineTypeA
instance variable names	inBuf
instance methods	

accessing

inBuf

```
^ inBuf
```

inBuf: aString

```
inBuf <- aString
```

simulation control

tasks

```
"Although this method looks very similar to the same method in
 the superclass MachineTypeA, it is actually necessary to
 reimplement it here. The difference between the methods is that
 the method described here must first acquire a resource before
 completing the rest of its tasks.The method described in the
 superclass did not perform this initial action."

| exponVarTime |
exponVarTime <-
            (Exponential mean: meanTBF) next
                        + Simulation active time.
"Must get the time of the simulation from the model"
[Simulation active time >
                  dependent simTime]
      whileFalse:
      [[Simulation active time > exponVarTime]
            whileFalse:
            [self inquireFor: 1 ofResource: inBuf]
                  whileFalse: [].
             self acquire: 1 ofResource: inBuf.
             "redisplay buffer at this point"
             self changed.
             self holdFor:
                  (Normal mean: cycleTimeMean
                   deviation: cycleTimeDev) next.
             self produce: 1 ofResource: outBuf.
             "redisplay buffer at this point"
             self changed]
       self holdFor:
             (Exponential mean: meanTtoRep) next.
       exponVarTime <-
                  (Exponential mean: meanTBF) next
                        + Simulation active time]
```

When the controller has all of the information about a machine, it passes it on to the model in the method, **createMachine: input: output: cycleTime: cycleTimeDev: mtbf: mttr: location:**. The model responds to this message by producing an instance of either **MachineTypeA** or **MachineTypeB**. It then adds this object to the collection of machines in the simulation.

The protocol description of the model is given next.

FactorySimulation instance protocol

menu messages

createMachine: machineName
input: inputBufferName
output: outputBufferName
cycleTime: cycleMean
cycleTimeDev: cycleDev
mtbf: failTime
mttr: repairTime
location: displayLoc

Create either an instance of MachineTypeA or MachineTypeB, and initialize the instance variables. Add the object to the database of machines.

createBuffer: bufferName
location: displayLoc

Add an entry, bufferName, to the database for buffers.

runSim: simTime

Start the simulation running, and have it run for the amount of time indicated by simTime.

When the controller sends the message runSim: to the model, the model must respond by initiating the simulation. This is accomplished by simply sending the message startUp to an instance of the class **Factory**, **which is a subclass of class** **Simulation**. Upon receipt of this message class **Factory** initializes and maintains the simulation until completion, at which point control is passed back to the model. The initialization of the simulation objects is done through the messages **defineArrivalSchedule** and **defineResources**. The implementation of these methods is shown next in the implementation description of class **Factory**.

```
class name            Factory
superclass            Simulation
instance methods

defineArrivalSchedule
      "loop through collection of machines in
            FactorySimulationModel"
      self scheduleArrivalOf: machine at: 0.0.

defineResources
      "loop through collection of buffers in
       FactorySimulationModel"
      self produce: 0 of: 'buffer'
```

The implementation description of the model is given next.

```
class name                    FactorySimulation
superclass                    Object
instance variable names       machines buffers simTime
instance methods

menu messages

createMachine: machineName
        input: inputBufferName
        output: outputBufferName
        cycleTime: cycleMean
        cycleTimeDev: cycleDev
        mtbf: failTime
        mttr: repairTime
        location: displayLoc
        "First determine if machineName is a MachineTypeA
         or MachineTypeB. Create the appropriate object,
         and add it to the ordered collection of machines."
        | tempMachine |
        (inputBufferName = nil)
              ifTrue:[tempMachine<- MachineTypeA new]
              ifFalse:[tempMachine<- MachineTypeB new.
                       tempMachine inBuf: inputBufferName].
        tempMachine outBuf: outputBufferName.
        tempMachine cycleTimeMean: cycleMean.
        tempMachine cycleTimeDev: cycleDev.
        tempMachine meanTBF: failTime.
        tempMachine meanTtoRep: repairTime.
        tempMachine location: displayLoc.
        tempMachine machineName: machineName.
        tempMachine addDependents: self
        machines addLast: tempMachine

runSim: simTime
        | sim |
        sim <- Factory new startUp.
        [sim time < simTime]
              whileTrue: [sim proceed].

update: aParam
        "The model receives this message when a machine object has
         modified the size of one of the buffers. The model must therefore
         inform the view that the model has changed, so the view can
         respond accordingly."
        self changed
```

VI.II. The Controller

The controller has command of the user menu options. The yellow button, by convention, is chosen to control the menu functions. Each option in the menu has a method corresponding to that option. Figure 2 showed the standard options available to the user with the yellow button. Shown below is the protocol description for the controller.

FactorySimulationController instance protocol

initialization	
initialize	Initialize instance variables. Set up the yellow button menu. Inform the view that the model has changed.
menu messages	
setYellowButtonMenu	Initialize the pop-up menu seen when the yellow button is depressed. This menu contains all of the options available to the user at that time.
addMachine	Prompt the user for all of the characteristics associated with the machine. Have the user place the machine in the display. Send a message to the model to add this machine to the simulation. Send messages to the model to add the buffers to the simulation. Reinitialize the user menu. Inform the view that the model has changed.

moveMachine	Have the user move a machine within the display. Inform the model of the machines new location. Inform the view that the model has changed.
moveBuffer	Have the user move a buffer within the display. Inform the model of the buffers new location. Inform the view that the model has changed.
modifyMachine	Allow the user access to the machines instance variables. Inform the model of the changes. Inform the view that the model has changed.
removeMachine	Send a message to the model to remove a machine from the simulation.
runSim	Prompt the user for the length of time which the simulation should run for. Send a message to the model to run the simulation for this length of time.
quitSim	Break down the MVC paradigm for factory simulation.
machine entry	
addData	Display a menu of the machine's characteristics. Do not allow the user to exit this menu until all of the machine's characteristics have been entered.
getMachName	Prompt the user for the name of the machine. If the name is already in use, prompt the user for a new name.
getInBufName	Prompt the user for the name of the input buffer.

getOutBufName	Prompt the user for the name of the output buffer.
getCycleTimeMean	Prompt the user for the mean cycle time of the machine.
getCycleTimeDev	Prompt the user for the cycle time deviation of the machine.
getMTBF	Prompt the user for the mean time between failures.
getMTTR	Prompt the user for the mean time to repair.
doneAdd	Verify that all of the characteristics for the machine are entered. If true, exit the addData menu. If false, do not allow the exit.

If the user chooses, for instance, to add a machine to the simulation. The yellow button menu option ***add machine*** is selected. By doing so, the corresponding method addMachine is sent to the controller. The controller responds to this message by prompting the user for all of the machines characteristics. It then directs the user to place the machine somewhere in the simulation window. The controller then passes all of the information it has gathered about the machine on to the model. After the machine has been added to the model's data base, the controller directs the placement of the input and output buffers associated with that machine. The buffers' locations are then passed on to the model so that they too can be added to the data base.

To prompt the user for the characteristics of the machine being added to the system, the controller displays the menu shown in figure 5. When the user chooses a selection, they are prompted for the correct value for that parameter. The user is not permitted to exit this menu until all of the characteristics have been entered. Therefore, a selection of ***completed entering data*** will do nothing until all of the other selections have been executed.

machine name =	*undefined*
input buffer =	*undefined*
outbut buffer =	*undefined*
cycle time mean =	*0.0*
cycle time dev =	*0.0*
mtbf =	*0.0*
mttr =	*0.0*
completed entering data	

Figure 5
Machine Entry Menu Options

When the user has completed entry of the machines and buffers to be used in the simulation, the menu option ***run simulation***, in figure 2, can be chosen. When this is done, the corresponding method, runSim, is sent to the controller. This method must first find out, from the user, the amount of time that the simulation will run. The controller then sends this information to the model in the message runSim:. The model is then responsible for starting the execution of the simulation. When the simulation is complete, the user will once again have the yellow button menu options shown in figure 2.

The implementation description of the controller is given next.

```
class name               FactorySimulationController
superclass               StandardSystemController
instance variable names  machName inBufName
                         outBufName cycleMean
                         cycleDev meanTBF meanTTR
instance methods

initialization

initialize
        super initialize.
        initialize instance variables
        self setYellowButtonMenu

menu messages

setYellowButtonMenu
        model has at least one machine
                ifTrue: [yellowButtonMenu:
                                (PopUpMenu labels: 'add machine\
                                        move machine\move buffer\
                                        modify machine\remove
                                        machine\run simulation\quit'
                                        withCRs)
                        yellowButtonMessages: #(addMachine
                                        moveMachine moveBuffer
                                        modifyMachine removeMachine
                                        runSim quitSim)]
                ifFalse: [yellowButtonMenu:
                                (PopUpMenu labels:
                                        'add machine\quit' withCRs)
                        yellowButtonMessages: #(addMachine
                                        quitSim)]
```

addMachine

```
"First prompt user for all characteristics associated with the
 machine being added. Have user place machine and buffers.
 Pass all info on to model."
| machLoc inBufLoc outBufLoc |
self addData.
place machine in display.
model createMachine: machName
        input: inBufName output: outBufName
        cycleTime: cycleMean cycleTimeDev: cycleDev
        mtbf: meanTBF mttr: meanTTR
        location: machLoc.
input buffer new
      ifTrue: [place buffer in display.
            model createBuffer: inBufName
                      location: inBufLoc].
output buffer new
      ifTrue: [place buffer in display.
            model createBuffer: outBufName
                      location: outBufLoc].
self setYellowButtonMenu.
model changed
```

runSim

```
prompt user for time for simulation.
model runSim: simTime.
self setYellowButtonMenu.
```

machine entry

addData

```
|addDataMenu labelString actions|
labelString <- WriteStream with:
'machine name = ', machName printString,
'\input buffer = ', inBufName printString,
'\output buffer = ', outBufName printString,
'\cycle time mean = ', cycleMean,
'\cycle time dev. = ', cycleDev,
'\mtbf = ',meanTBF,
'\mttr = ', meanTTR,
'\completed entering data'.

actions <- #(getMachName getInBufName
             getOutBufName getCycleTimeMean
             getCycleTimeDev getMTBF getMTTR
             doneAdd).
addDataMenu <- PopUpMenu
                 labels: (labelString contents)
                                        withCRs
                 lines: #(1 2 3 4 5 6 7).
^self perform:(actions at: (addDataMenu
            startUpAndWaitForSelectionAt: aPoint))
```

getMachName

```
[machine name is undefined or
machine name is already in use]
whileTrue: [
  FillInTheBlank
      request: 'enter the name of the machine'
      displayAt: Sensor cursorPoint
      centered: true
      action:[:response| ]
      initialAnswer: machName printString.
  machName <- response].
^self addData
```

getInBufName

```
FillInTheBlank
  request: 'enter the name of the input buffer'
  displayAt: Sensor cursorPoint
  centered: true
  action:[:response| ]
  initialAnswer: inBufName printString.
inBufName <- response.
^self addData
```

getOutBufName

```
FillInTheBlank
  request: 'enter the name of the output buffer'
  displayAt: Sensor cursorPoint
  centered: true
  action:[:response| ]
  initialAnswer: outBufName printString.
outBufName <- response.
^self addData
```

getCycleTimeMean

```
FillInTheBlank
  request: 'enter the machine's mean cycle time'
  displayAt: Sensor cursorPoint
  centered: true
  action:[:response| ]
  initialAnswer: cycleMean.
cycleMean <- response asNumber.
^self addData
```

getCycleTimeDev

```
FillInTheBlank
    request: 'enter the machine's cycle time
              deviation'
    displayAt: Sensor cursorPoint
    centered: true
    action:[:response| ]
    initialAnswer: cycleMean.
cycleDev <- response asNumber.
^self addData
```

getMTBF

```
FillInTheBlank
    request: 'enter the machine's mean time
              between failures'
    displayAt: Sensor cursorPoint
    centered: true
    action:[:response| ]
    initialAnswer: meanTBF.
meanTBF <- response asNumber.
^self addData
```

getMTTR

```
FillInTheBlank
    request: 'enter the machine's mean time to
    repair'
    displayAt: Sensor cursorPoint
    centered: true
    action:[:response| ]
    initialAnswer: meanTTR.
meanTTR <- response asNumber.
^self addData
```

VI.III. The View

The primary objective of the view is to keep the MVC triad together. This is accomplished by establishing the MVC triad in the view where methods are inherited that connect the view with its model and controller.

The view's other responsibility is to display the data contained in the model. It is therefore necessary for the model to inform the view when a change in the data occurs.

The protocol description of the view is shown next.

FactorySimulationView instance protocol

initialization	
startSim	Establish the MVC triad. Make the window active.
displaying	
displayView	Display the data in the model.
update: aParam	Inform the view that the data in the model needs to be redisplayed.

Following is the implementation of the view.

class name FactorySimulationView
superclass StandardSystemView
instance methods

startSim

```
insideColor <- Form white.
borderWidth <- 2.
borderColor <- Form black.
self model: FactorySimulation new
    controller: FactorySimulationController new.
window <- model window
self label: 'Factory Simulation'.
controller open
```

displayView

```
| trans |
super displayView.
"display the model"
trans <- WindowingTransformation
                window: window
                viewport: insetDisplayBox.
(model allObjectsToBeDisplayed)
            do: [:each | (trans applyTo: each)display]
```

update: aParam

```
self displayView
```

VII. Conclusion

The purpose of this paper was to present a high level design for an object-oriented, menu driven front-end of a simulation package for simulating automated flow lines. Using the abstract tools provided by Smalltalk-80 for simulation, classes were established to represent the machines and storage buffers in an automatic flow line.

The main goal in designing this front end was to develop an interface that is easy to use. This was accomplished by the use of menus. The user selects the next action from the main menu and is then either presented with another menu of choices or is prompted for specific information.

This paper presented the structure for the interface design. In some cases, specific detail was presented to fully describe the Smalltalk-80 implementation. In other cases, a partial description was provided in order to give the reader the basic flow of the design.

It would be desirable to integrate the menu-driven front-end with the SimTalk package that was mention earlier. This paper used two classes of objects to be simulated. Each was a subclass of the abstract class **SimulationObject** provided in Smalltalk-80. By making these classes subclasses of **SimulationObject**, they inherited the basic structure of any object in a simulation. In order to combine this with the SimTalk package, classes **MachineTypeA** and

MachineTypeB would need to be subclasses of a more robust class provided by SimTalk. In this way, the objects not only inherit the basic structure for simulation, they would also inherit the more advanced features provided by SimTalk.

By changing the superclass of the machine objects, the user interface would require very little modification to be used with the SimTalk package. Enhancements could be made to prompt the user for other choices that SimTalk provides, such as what simulation or output package is desired. Due to Smalltalk's modularity, the front-end designed here could be added to the SimTalk package with little or no modifications to the SimTalk package itself.

The combination of this menu-driven front-end and the package provided by SimTalk would result in a "user oriented" simulation package for manufacturing system design.

BIBLIOGRAPHY

Doe, Lawrence Whittier, *Object Oriented Simulation of Manufacturing Systems: A Smalltalk Experience*, Dissertation, Lehigh University, 1989

Goldberg, Adele and David Robson, *SMALLTALK-80: The Language*, Addison-Wesley, Reading, Ma, 1989

Grant, F. Hank, "Simulation in Designing and Scheduling Manufacturing Systems", *Design and Analysis of Integrated Manufacturing Systems*, National Academy Press, Washington, D.C., 1988, 134-147

Groover, Mikell P., *Automation, Production Systems, and Computer-Integrated Manufacturing*, Prentice-Hall, Englewood Cliffs, NJ, 1987

Pinson, Lewis and Richard Weiner, *An Introduction to Object-Oriented Programming and SMALLTALK*, Addison-Wesley, Reading, Ma, 1987

Vita

Pamela S. Woodbury was born April 7, 1963 in Cuba, New York. In 1987 she graduated Cum Laude with a Bachelor of Science in Electrical Engineering from the University of Miami, Miami, Florida.

Currently she is employeed at AT&T Bell Laboratories in Allentown, Pa. where she works in a software development group for test program generation.