CptS 464/564 Project #3

Assigned: Tuesday, March 25, 2003
Due: **Tuesday, April 8, 2003 (start of class)**
Weight: 10% of Final Grade

Overview & Objectives

You have been asked to instrument a Widget Factory. Your 'Manager' has heard that CORBA is 'cool' and that the OMG Event Service is even 'cooler', so you must use events and events channels to complete your assignment. Many models of widgets can be produced at the widget factory and each widget is assigned an identifying number. Several assembly lines exist in the factory and each one can produce widgets of varying colors. Some widgets take longer than others to produce, but each time a widget is produced, an event is sent to the event channel. A WidgetFactory program consumes these widgets (and all other events) from the event channel. Statistics are kept on widget production and a factory store sells widgets.

Project #3's objectives include:

- Gain more CORBA experience
- Gain an understanding of and experience working with the OMG Event Service
- Learn how to co-locate multiple servants within a single program
- Learn how to use the CORBA ANY type.

Resources that might be of use to you include:

- Messages that will be sent to cs564@eecs.wsu.edu (the class e-mail list)
- /net/niflab/orbacus/cs564/Project3/Widget (Class supplied IDL files, etc.)
- /net/niflab/orbacus/OB-4.1.2/event/demo

The Widget Factory

A diagram of the completed Widget Factory and all of its parts is shown in Figure 1.

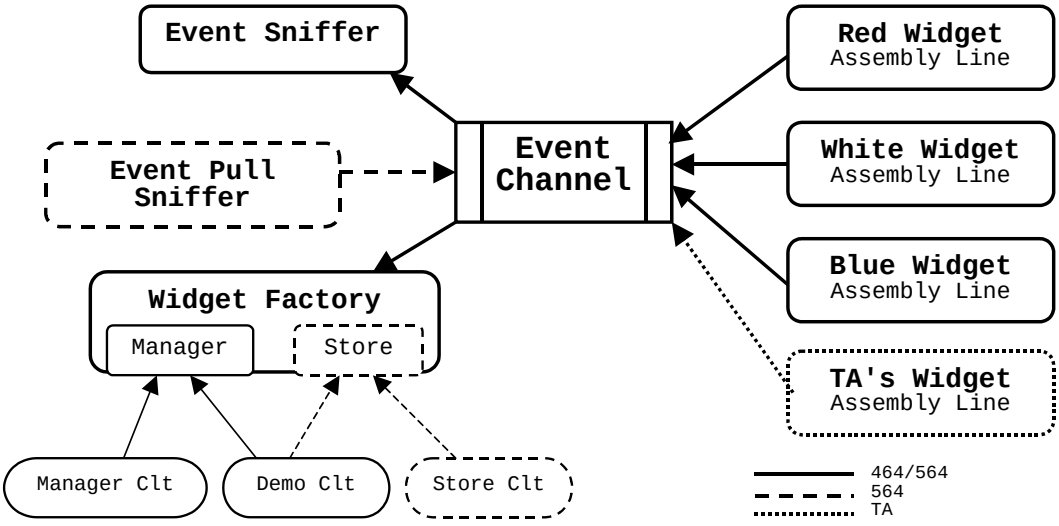


Figure 1: The Widget Factory

On the right of Figure 1 are the ***event suppliers*** (aka the widget assembly lines). The red, white, and blue widget assembly lines are to be implemented by you, the TA will implement a fourth assembly line. The different colors of widgets take different amounts of time to produce: 3 seconds per red widget, 5 for white, and 4 for

blue. Each of the red, white, and blue assembly lines should operate at full capacity (e.g., the blue widget assembly line will send an event to the event channel every four seconds). The TA's widget assembly line is 'magic' and it can produce widgets of any color and at a variable rate from 1 to 5 seconds per widget. The TA's assembly line will also produce events consisting of string values and CORBA::long values. All of the event suppliers will operate in 'Push' mode**. Your widget events must have the widget model set to your EECS user name.** The TA's widget assembly line will produce 'ta564' widgets.

On the left side of Figure 1 are the ***event consumers***. CptS 464 students are required to implement an event sniffer and a widget factory program, both of which will operate in 'Push' mode. The event sniffer must capture and print out every event received regardless of its type (i.e., strings, longs, widgets). Graduate students must also implement a 'Pull' mode event sniffer. The widget factory program is special because it contains two co-located servants, an event consumer servant and a factory manager servant. CptS 564 students will also need to implement a third co-located servant that sells widgets.

At the bottom left-hand side of the Figure 1 are two ***client programs*** that must be implemented (three for graduate students). The manager and store clients communicate with their respective servants. These client programs should have a (very) simple user interface. The demo client program will have no user interface because its purpose is to automatically test every IDL method call.

Before proceeding further, the IDL for widgets and the widget factory needs to be introduced.

Widget.idl

```
// Widget.idl
// CptS 464/564 -- Project #3 – Spring 2003
// This file defines a widget structure used to identify widgets produced
// by a factory
#ifndef _WIDGET_IDL_
#define _WDIGET_IDL_
enum ColorType { C_RED, C_WHITE, C_BLUE };  // widget colors
typedef long IdType;                // id number for widgets
struct Widget {
  string    model;
  IdType    id;
  ColorType color;
};
#endif
```

WidgetFactory.idl

```
// WidgetFactory.idl
// CptS 464/564 -- Project #3 – Spring 2003
// This file defines a widget factory. The factory has a manager in charge
// of shift changes and factory statistics. Graduate students also need to
// implement the factory's store.
//
#ifndef _WIDGETFACTORY_CS564_IDL_
#define _WIDGETFACTORY_CS564_IDL_
module WidgetFactory {
  // This will be a widget factory--so we'll include the widget types
  //-----
  #include "Widget.idl"
  struct Stats {
    // Stats is designed to report statistics about Widgets produced by this
    // factory.
    //-----
    long  eventCnt;     // total number of ALL events received
    string model;       // name of the model for which stats are being reported
    long   redCnt;      // count of red widgets produced (events received)
    long   whiteCnt;    // count of white widgets produced
    long   blueCnt;     // count of blue widgets produced
    double elapsedTime; // elapsed time (used to compute averages)
  };
  exception NoStatsInfo {
    // If stats are requested for widget model for which no stats exist then
    // this exception is thrown. This occurs when the model specified does
    // not exist or when stats were not accumulated for the given model.
    //-----
    string reason;
  };
  exception OutOfStock {
    // If an order is placed for an out of stock item, throw this exception
    //-----
    string reason;
  };
  interface Manager {
    // Provide a management type object to the factory
    //-----
    void
    ShiftChange(in string shiftManager);
    // At a shift change, the shift managers name is updated and the current
    // stats are reset to zero for the start of this shift
    //-----
    void
    GetShiftStats(in string model, out Stats currStats)
    raises (NoStatsInfo);
    // Get the stats for the current shift
  };
  //-----GRADUATE STUDENTS ONLY BEYOND THIS POINT-------
  interface Store {
    // Widgets can be ordered from the Factory via its Store interface
    //-----
    long
    GetWidgetInventory (in string model, in ColorType color);
      // The inventory for a given widget model and color is returned.
      //-----
    long
    BuyWidget (in string model, in ColorType color, in long quantity)
    raises (OutOfStock);
      // Buy 'quantity' widgets of a given model and color. Return the number
      // of bought widgets (may be less than 'quantity' if stock is limited,
      // but it must always be greater than zero). Raise OutOfStock if none
      // of the specified widgets are in stock at the factory.
  };
};
#endif
```

The ***manager client*** must be able to invoke the 'ShiftChange' and 'GetShiftStats' methods. Your manager servant must track statistics for at least two different widget models. Naturally the required model values will be

'ta564' and your EECS user name. Statistics for other widget model values may be discarded. The manager client must also track the total number of events that it receives from the event channel.

The ***store client*** must be able to invoke the 'GetWidgetInventory' and 'BuyWidgets' methods. You must keep an inventory of your own widget model plus the TA's model. Every time an event arrives you should increase the inventory by one. Each time a widget is bought, the inventory will decrease by the quantity ordered.

The ***demo client*** is an automated (testing) tool that will demonstrate the key functionality of the Widget Factory code that you've written. It must execute the following steps:

1. Obtain object references for the Manager (& Store) servants (via the Naming Service)
2. GetShiftStats ('ta564', currStats)
3. GetShiftStats ('No-Such-Model', currStats)
4. ShiftChange ('Your-User-Name')
5. wait 60 seconds
6. GetShiftStats('Your-User-Name, currStats)
7. GetShiftStats('ta564', currStats)

—Graduate Students Only—

8. GetWidgetInventory('No-Such-Model', blue)
9. GetWidgetInventory('ta564', white)
10. BuyWidget('ta564', white, 5)
11. repeat step 10. until an exception is thrown
12. GetWidgetInventory('Your-User-Name', red)
13. wait 5 seconds
14. GetWidgetInventory('Your-User-Name', red)

You must submit a printout of this program's execution. During the demo you will be asked to run this program. Once it has run, you must then justify / explain why you think its results are correct. (Note/Hint: Be prepared for the TA's widget assembly line parameters to change—there will be no guarantee that the TA assembly line will produce the name number/color of widgets during the development/testing phase as it will during your demo).

Additional Work for CptS 564 Students

Two of the additional tasks that CptS 564 students must do have been mentioned previously. These tasks are to implement a 'Pull' event sniffer and to implement a store servant and client. The third task is to consider the OMG Notification service as an alternative to the event service: in class we briefly mentioned the concept of event filtering. You need to learn more about the OMG Notification Service and then explain how this Widget Factory assignment could be modified to use the Notification Service. You must then also explain why using the Notification Service would or would not be an improvement (or could it be both?). This discussion should be limited to a 1/2 page (single-spaced) or less.

Simplify, Simplify, and Simplify

The point of this programming assignment is to work with CORBA. To that end you may make as many simplifying assumptions as you would like as long as you can still meet all of the given design constraints. For example, you can decide if you want to make three separate cut-n-paste copies of the red, white, and blue widget assembly line programs or you may make one assembly line executable that sets the widget color and timing parameters via command line arguments.

Grading

Turn in hardcopy of all your created files. Note, the “created” files include only those files that you've created/edited yourself. You must also turn in printouts showing the run-time execution of your project’s executables. These printouts must show/verify that your event suppliers and consumers work and that every method and exception specified in the Factory.idl file is implemented. One method of obtaining these run-time printouts is to embed print statements in your code. For example, every event supplier (or consumer) should print out a notification message every time an event is generated (or consumed).

All Pullman students will be expected to demo your application to a TA. During the demo you will be asked to run your code and to explain it as well. The course syllabus states "DO YOUR OWN WORK!" so naturally you should be able to explain all of the code you've written and most of the code that you've copied from the example programs (at least you should know what the copied example code is doing in a general sense). Demo times and arrangements will either be mentioned in class or via the class e-mail list. Your demo is timed—you will have points deducted if you exceed your 10-minute time slot. Please rehearse your demo before hand and come prepared to answer simple questions about your project.

Finally, do not modify or otherwise change the last accessed dates on your created files after the submission date, in case we need to check the files. These final files should reside in the directory /net/niflab/<yourusername>/project3. **(Note that this location is different from what was required for the first two projects!)**

Grading Criteria (Penalty points listed inside parenthesis)

Project Writeup

10/15[g] pts	Cover Sheet	
	2	Your Name
	2	Code Location
	3	One (1) positive comment about the project (no more than a paragraph long!)
	3	One (1) suggestion about improving the project (no more than a paragraph long!)
	0/5[g]	OMG Notification Service comments (1/2 page or less!)
10/13[g] pts	Run-Time Printouts	
	5	Event Sniffer output
	5	Demo Client output
	3[g]	Pull Event Sniffer output
10 pts	Source Code (Files may be in any order as long as DemoClient.cpp is first)	
	5	Labeled
	5	Complete
	(-5)	DemoClient.cpp not included as the first source code printout

Demonstration / Working Code

20/22[g] pts	Makefile & Running Programs	
	4	Makefile
	6	[Red | White | Blue] WidgetSuppler program(s)
	2	EventSniffer
	1[g]	EventPullSniffer
	2	WidgetFactory
	2	ManagerClient
	1[g]	StoreClient
	4	DemoClient
12 pts	Event Suppliers	
	5	Connect to Event Channel
	3	Supply widgets (red, white, blue)
	2	Correct production rates (3, 5, 4)
	2	Widget Ids increment
	(-5)	Incorrect widget model value
15 pts	Event Sniffer	
	9	works
	2	CORBA::long sniffed
	2	string sniffed
	2	Widget sniffed
0/7[g] pts	EventPullSniffer	
	5[g]	works in 'Pull' mode
	2[g]	events sniffed
18/21[g]	WidgetFactory	
	5	works
	5	EventConsumer servant
	2	Manager servant
	6	Correct statistics computed
	1[g]	Store servant
	2[g]	Correct inventories kept
5 pts	ManagerClient	
	1	works
	2	calls ShiftChange
	2	calls GetShiftStats
0/5[g] pts	StoreClient	
	1[g]	works
	2[g]	calls GetWidgetInventory
	2[g]	calls BuyWidget

Penalty Points

(-1..-10)	Exceed Demo time (10 minutes)
(-10) pts	Use of global variables
(-20) pts	Demo not performed in the SNIF lab (ETRL 301) **on** a SNIF machine
(-2..-20) pts	Event suppliers are left running after you log off. (You will be warned the first time, the second time will cost your 2 points, after that you will lose 5 pts per time that I 'kill' your leftover tasks.)

Note: [g] denotes points for a CptS 564 task

Note: You will receive only partial credit if the correct working of your code cannot be verified by watching screen printouts when your DemoClient program runs—you ***must*** embed print statements if you want full credit!

Implementation Notes

The event service we will use is part of the ORBacus system, and is located at /net/niflab/orbacus/bin/eventserv. You should each run your own instance of the event service. You will need to be able to find the event service from your programs. The easiest way I know to do this is to start the event server with the --ior flag, causing it to write its IOR on standard output. Redirect standard output to a file, thusly:

```
/net/niflab/orbacus/bin/eventserv --ior >Event.ref
```
Then, when you start your programs use the backquote technique to incorporate the contents of file Event.ref on the command line:
```
yourprogname -ORBInitRef EventService=`cat Event.ref`
```
Within your program you can use resolve_initial_references("EventService") to get the handle you need for the event service.

We will use the so-called untyped event service, the default, and we will not use the ORBacus proprietary Event Channel Factory feature.

The OMG Notification service

ORBacus does not provide a Notification service. To write your discussion about the Notification service you will need to refer to the specification at http://www.omg.org/docs/formal/02-08-04.pdf. The specification is over 200 pages – I don't expect you to read it in detail, but concentrate on the overview: try to understand how the Notification service goes beyond the features of the Event service and solves some problems that were discovered in the specification of the Event service.