Donald G. Firesmith

Advanced Software Technology Specialists, 2910 Webster St., Fort Wayne, IN 46807; email: 73664.3515@compuserve.com.

Using parameterized classes to achieve reusability while maintaining the coupling of application-specific objects

TO MAXIMIZE REUSABILITY, classes should be as independent of other classes as is practical. However, their instances often must be coupled with instances of other classes if they are to collaborate to provide some application-specific required capability. This article compares three object-oriented designs for a common embedded, real-time mechanism and shows how parameterized classes (a.k.a. generics in Ada and Eiffel, templates in C++) achieve reusability while retaining the necessary coupling of their instances.

EXAMPLE MECHANISM

A mechanism is a reusable pattern of collaborating objects or classes that supply some useful capability. A common mechanism in real-time embedded applications is a control loop involving a substance, a sensor, an actuator, and a gauge. Some attribute of the substance, such as temperature or pressure, is read by the sensor. Once the raw value from the sensor is converted into appropriate units, the attribute is used to control the actuator that in turn maintains the value of the substance's attribute within some desired range. The value of the attribute is also displayed on the gauge so that it may be observed by users of the mechanism. The software application implementing this mechanism is hosted on some microprocessor that communicates with the actuator, substance, gauge, and sensor via multiplexing analog to digital and digital to analog converters.

Using the notation of the ASTS Development Method (ADM),[1] the context diagram for this mechanism is documented in Figure 1. In this context diagram, a hardware–system object is drawn as a three dimensional box, a person–system object is drawn as a circle, and a subassembly of software objects and classes is drawn as a rounded rectangle. The icons for classes consist of two overlapping icons of the corresponding objects. Icons for objects and classes that contain software are drawn with thick borders.

The most straightforward object-oriented design would be to have one software object model each system object so that when individual system objects are upgraded or replaced, only a single software object would have to be changed. The control loop that reads the sensor and controls the actuator to maintain the value of the substance's attribute is independent of any other mechanism running on the microprocessor. The software object modeling the substance should therefore be concurrent, because one of its operations implements this logically concurrent control loop . Most real-time applications contain multiple mechanisms with their own threads of control that share the two converters. If the software objects modeling these two converters are concurrent and ensure mutually exclusive access via their own threads of control, then they can ensure that their attributes are not corrupted by the interleaving of their operations due to time slicing. For the same reason, the software object modeling the microprocessor should be concurrent to ensure mutually exclusive handling of the messages from the converter objects. The software objects modeling the actuator, gauge, and sensor may remain sequential because they are encapsulated within the body of the mechanism subassembly, and the Substance object provides their single thread of control via message passing.

Using ADM notation, the semantic net for the object architecture implementing this mechanism is documented in Figure 2. These semantic nets document concurrent software objects as parallelograms and sequential software objects as rectangles. Objects drawn on the border of a subassembly are exported and therefore visible to other objects and classes outside the subassembly, whereas objects drawn nested inside the border of a subassembly are hidden within the body of that subassembly and are not accessible by other software outside the subassembly.

The client-server relationship between the software objects in Figure 2 is the reverse of the client-server relationship between the system objects in Figure 1. This mirror image is necessary to ensure proper flow of control and is one of the ways that the structure and behavior of system and software objects and classes differ. This is the reason that ADM incorporates separate semantic nets and interaction diagrams at the system and software levels and uses different icons for system and software objects and classes.

EXAMPLE CLASS ARCHITECTURE THAT COUPLES THE CLASSES

The most straightforward design would be to have the architecture of the classes mirror the architecture of their instances. Fig-

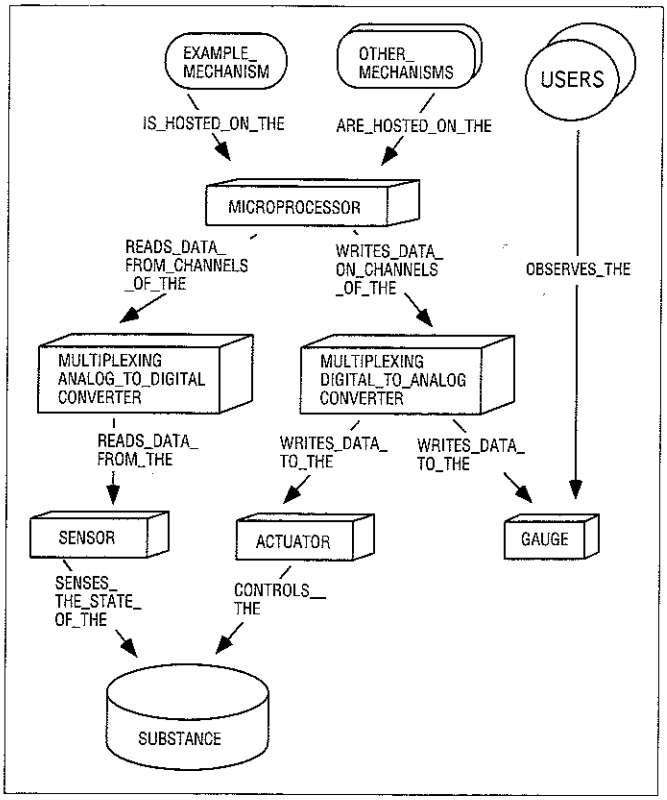

Figure 1. ADM context diagram for the example mechanism.

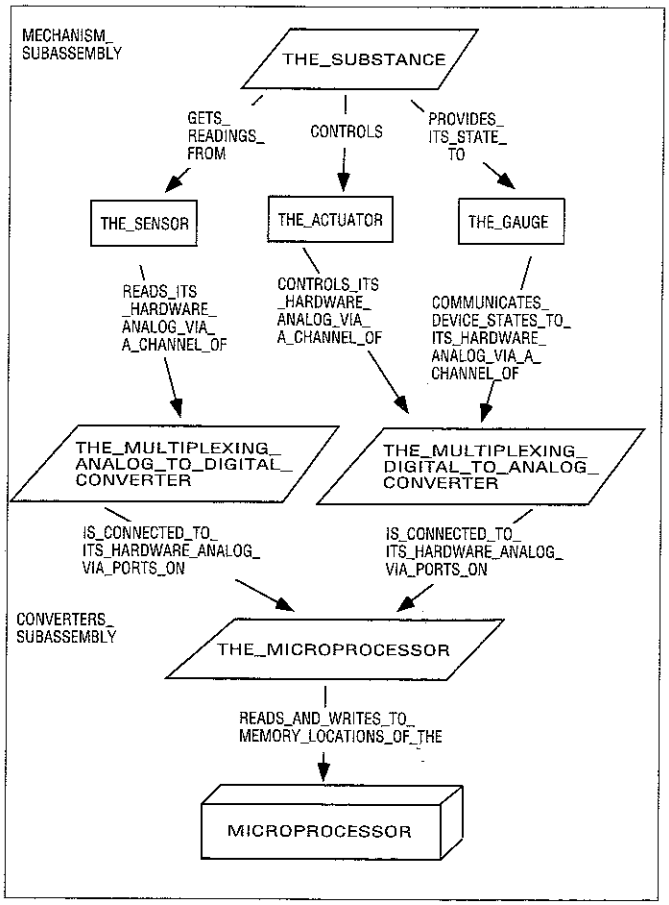

Figure 2. ADM software semantic net showing the object architecture.

ure 3 provides an ADM software semantic net documenting the resulting class architecture, which integrates the instances by coupling the classes. Although this architecture will work for this specific mechanism, it also hardwires the classes together so that they are not reusable in different mechanisms or on different variants of this mechanism. For example, substances may have more than one sensor or actuator. Also, not every substance displays the value of its attribute(s) on some gauge. For example, the gauge may display sensor values directly without regard to the substance being measured by the sensor. For these reasons, using traditional classification techniques does not provide the necessary flexibility. Although inheritance can supply this required flexibility by permitting developers to create a subclass for each variant actuator, substance, gauge, and sensor, it also involves significant redundant development if the number of variant classes is large while the number of instances per variant is small.* This design is therefore not optimal.

EXAMPLE HYBRID OBJECT ARCHITECTURE THAT DECOUPLES THE CLASSES

Figure 4 documents a hybrid functional and object-oriented design that decouples the classes by pulling the control loop out of the Substance object where it belongs and objectifying the corresponding operation as a Control Loop object. This Control Loop object is concurrent because it encapsulates the logically independent concurrent control loop operation. It is also an example of the control object recommended by the Object-Oriented Software Engineering (OOSE) and Objectory methods.[2] Another common variant of this approach is to divide the Substance object into two pieces: a smart application-specific Substance that encapsulates the control loop and a dumb reusable Substance that only models the system aspects of the substance without knowledge of how it is to maintain the values of its attributes via control loop(s).

This approach clearly decouples the Substance object from the Actuator, Gauge, and Sensor objects and therefore makes all four objects more reusable. Unfortunately, it also has many disadvantages. Control loops are subject to frequent changes in requirements, and this approach mislabels a volatile function as an object.† The control loop is tightly coupled with the four decoupled objects. Whereas fan-in should be maximized because it promotes reuse, fan-out should be minimized because it increases data and control coupling and decreases maintainability. The Control Loop object knows too much about the implementation of the Substance object because it is now responsible for maintaining the substance's state within the proper range via the actuator. This software design does not mirror the system design. The second variant of this approach also violates the premise of object orientation that all resources of an object should be localized and encapsulated within that object. For these reasons, this approach is also not optimal and should be avoided.

* Although inheritance can accomplish everything that parameterization can and the reverse is not true, parameterization offers more convenient ways of doing certain things, which is why languages such as Ada 9X, C++, and Eiffel support both inheritance and parameterization.

† As requirements change, functions tend to be more volatile than data, which are more volatile than objects, which are more volatile than concrete classes, which are more volatile than abstract and generic classes.

RECOMMENDED DESIGN USING PARAMETERIZED CLASSES

Parameterized classes are recommended for creating decoupled reusable classes while retaining the coupling of their instances for collaboration. This recommended approach is documented in Figure 5: parameterized classes are drawn with dashed lines. For example, consider the parameterized SUBSTANCES_WITH_CONTROL_LOOP subclass. It is derived from a simple SUBSTANCES superclass to (1) encapsulate the control loop as an application-specific generic parameter and (2) decouple it from the ACTUATORS, GAUGES, and

Parameters supply flexibility, allowing classes to utilize application-specific control loops and servers.

SENSORS classes by making generic parameters out of the operations that would call instances of these server classes. Figure 6 is a white-box interaction diagram (WID) that documents the internal architecture of an instance of the SUBSTANCES_WITH_CONTROL_LOOP class. Assume that this class is part of an inheritance hierarchy. After flattening this inheritance architecture to make the parameterization of the parameterized SUBSTANCES_WITH_CONTOL_LOOP class clearer, we see that it is parameterized with four operations, three of which are used to decouple it from its server classes. See the following resulting specification of the parameterized class written in Version 2 of the OOSDL specification and design language[1]:

```
class SUBSTANCES_WITH_CONTROL_LOOP
  definition
    abstraction
      A substance with a single control loop maintaining and displaying
      the value of one of its attributes.
    end abstraction;
    responsibilities
      R1: Determine attribute value by reading sensor;
      R2: Calibrate attribute value;
      R3: Maintain attribute value using the actuator;
      R4: Display the attribute value on the gauge;
      R5: Change and return the state;
    end responsibilities;
  end definition;
  class interface
    message construct (Name);
  end class interface;
  instance interface
    parameter class ATTRIBUTES;
    class STATES is (Disabled, Calibrated, Malfunctioning, Uncalibrated);
    message calibrate_sensed_attribute
      (Calibration_Value : ATTRIBUTES)   is synchronous
    message destruct;
    message set (The _State : STATES)    is synchronous;
    message state_of return STATES       is synchronous
    exception FAILURE_OCCURED_IN;
  end instance interface;
  class implementation
    constructor construct (Name);
  end class implementation;
```

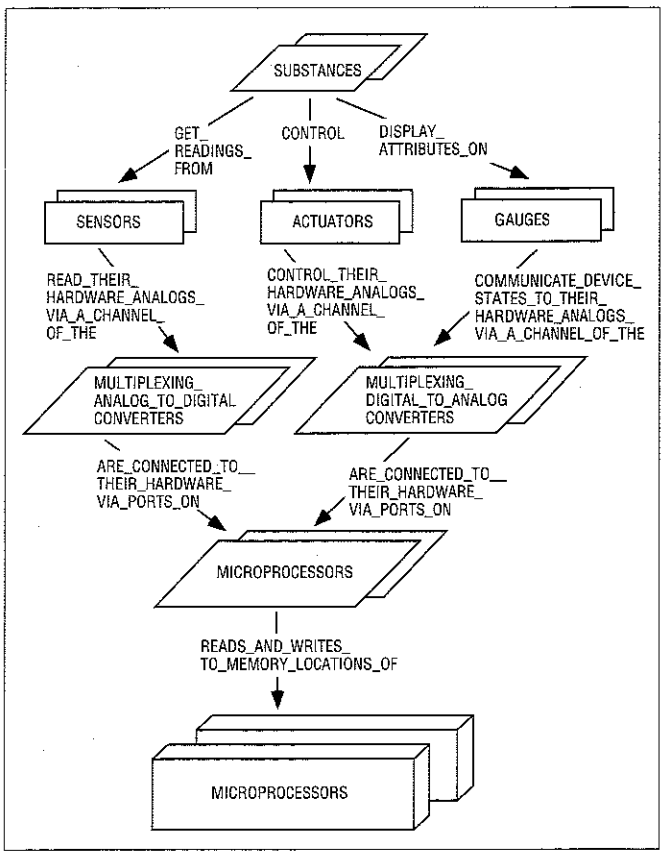

Figure 3. Initial class collaboration architecture with coupled classes.

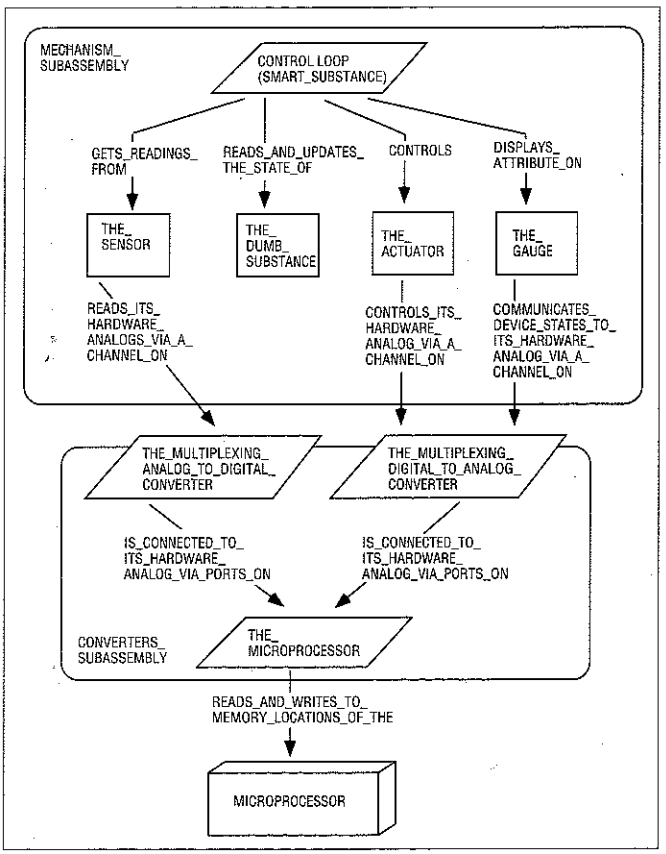

Figure 4. Object collaboration architecture using a functional control object.

Parameterized classes

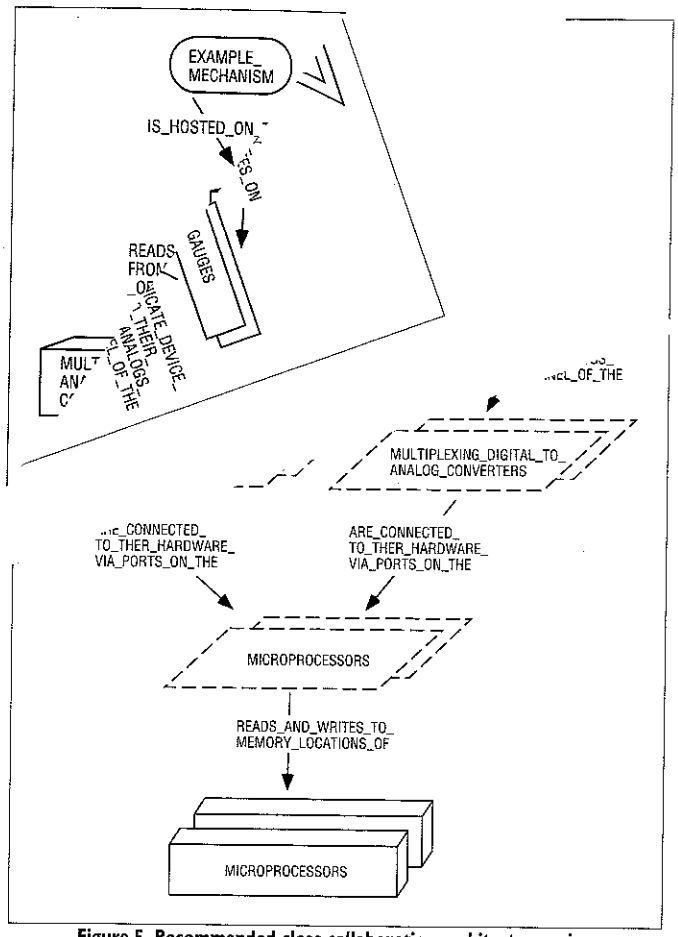

Figure 5. Recommended class collaboration architecture using parameterized classes.

```
interface implementation
  -- attributes
  variable Calibration_Factor   :REALS        is protected;
  variable Calibration_Value    :REALS
  variable New_State            :STATES
  variable The_Current_State    :STATES       is protected;
  variable The_Current_Value    :ATTRIBUTES   is protected;
  -- operations
  modifier accept_message                     is concurrent;
  parameter modifier calibrate;
  parameter modifier control_actuator; --decouples class from server
  destructor destruct;
  parameter preserver display;          --decouples class from server
  parameter preserver maintain_current_values  is concurrent;
  parameter modifier read_sensor;       --decouples class from server
  preserver return_state return STATES;
  modifier set (The_State : STATES);
end interface implementation;
end class; -- SUBSTANCES_WITH_CONTROL LOOP
```

The following benefits result from performing operations that send messages to the generic formal parameters of server objects :

- Instances of parameterized classes can be coupled so that they may collaborate to supply required capabilities, without the parameterized classes themselves being coupled.
- Parameters supply flexibility, allowing classes to utilize application-specific control loops and servers.

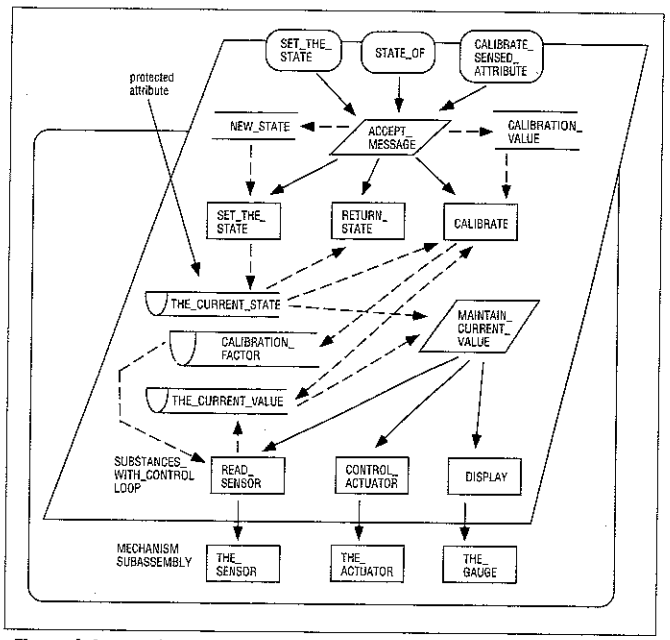

Figure 6. Interaction diagram of the Substance_With_Control_Loop object.

- Parameterized classes are more reusable and maintainable because no changes need be made in their design or code when the control loops or servers change.
- The designers of parameterized classes can mandate that specific operations be supplied before instantiation, because parameterized classes are abstract classes that list deferred resources as generic formal parameters.
- Parameterized classes are often easier to develop and use than nongeneric deferred classes because developers only need to concentrate on, develop, and reuse operations as generic parameters rather than develop entire subclasses.
- This approach simplifies the inheritance hierarchy, making it easier to understand and navigate, especially if CASE tool support is limited.

CONCLUSION

When parameterized with volatile control loop operations and operations that call server objects, parameterized classes are highly reusable without limiting extensibility as these control loops and servers change. They also avoid the objectification of control-loop functions by encapsulating control loops as operations in the bodies of the objects they control. This approach simplifies the design by limiting the number of objects in the design and simplifies the inheritance hierarchy by limiting the number of subclasses. In addition, it allows the software design to better mirror the hardware design and limits the amount of control and data coupling due to fan-out. ■

References

1. Firesmith, D.G. Object-Oriented Requirements Analysis and Logical Design: A Software Engineering Approach, Wiley, New York, 1993.
2. Jacobson, I., et al. Object-Oriented Software Engineering: A Use Case Driven Approach, Addison-Wesley, Wokingham, UK, 1992.