Appears in the *Journal of Object-Oriented Programming*, 11(7): 25-35, SIGS Publication Inc., Nov/Dec 1998.

Gary M. Weiss and Johannes P. Ros

AT&T Labs, Middletown, NJ
{gmweiss, hros}@att.com

Implementing Design Patterns with Object-Oriented Rules

ABSTRACT

ANSWER-4ESS, an operation support system for maintaining and monitoring the 4ESS switches distributed in the AT&T network, recently became the first product to use the R++ language—an extension to C++ that adds support for object-oriented (OO) rules. This article shows how these rules not only implement the intelligent behavior of an expert system, but also facilitate inter-object communication and enable frequently occurring design patterns to be implemented in a way that leads to more understandable, maintainable, and reusable code.

Most of AT&T's long-distance traffic is controlled by the 140 4ESS switches distributed throughout the AT&T network. Automatic Network Surveillance with Expert Rules (ANSWER) is the operation support system responsible for monitoring and maintaining these switching systems. ANSWER's main purpose is to perform alarm filtering: It takes as input alarms from the 4ESS switches and outputs an alert once it has identified a problem that requires human intervention. ANSWER's intelligent decision making and diagnostic reasoning is controlled by its expert system—the first production software written in R++, an extension to the C++ language that adds object-oriented rules. Previous work has focused on how R++ was used to implement model-based reasoning in ANSWER, by allowing each 4ESS to be represented as a hierarchical collection of devices.[1-3]

THE R++ LANAGUAGE

R++ is an extension to the C++ language that adds object-oriented rules.[4-5] It was developed by AT&T Labs research with the ANSWER project in mind—so that it and other systems could use rules and still take advantage of object-oriented technology and existing C++ code. R++ rules are considered another type of C++ member function and share the object-oriented properties of C++ member functions: inheritance, polymorphism, and dynamic binding. Listing 1 shows the declaration of class Person and two member rules for that class, called spouse_check and older_child. Note that *children* is a set of pointers to class Person. The optional *monitored* keyword in the class declaration identifies data members that may trigger rule evaluation.

Listing 1: Declaration of class Person

```
class Person {
   private:
   String                              name;
   monitored int                       age;
   monitored Person                    *spouse;
   monitored Set_of_p<Person>          children;

 rule spouse_check;
 rule older_child;
};
```

Rules have a special *if-then* syntax, where the if (antecedent) and then (consequent) parts are separated by an arrow (⇒). The rule in Listing 2a can be translated as: *if a person's spouse's spouse is not equal to the person then print an error*. R++ also provides the ability to write rules on container classes, like sets and lists. Using an R++ feature called "branch binding", a rule can be applied to each element of the container (the at-sign is the branch-binding operator). The rule in Listing 2b can be translated as: *if a person's child (for each child in the set children) is older than the person, then print an error*. Thus, in the example, if a person has 5 children and 2 of them are older than the parent, then two errors will be printed. R++'s ability to support rules on container classes facilitates inter-object communication and maximizes code generation (no code was needed to actually iterate through the set of children in 2b).

Listing 2: Two sample R++ rules

2a.

```
// Is my spouse married to me?
rule Person::spouse_check
{
 Person *sp = spouse † &&
 sp→spouse != this
⇒
 cout << "Error" <<sp→name
      << "married to"
      << sp→spouse→name;
};
```

2b.

```
// Is my child older than I am?
rule Person::older_child
{
 Person *child @ children  &&
 child→age > this→age
⇒
 cout <<"Error"<<child→name
      << " older than parent"
      << this→name << endl;
};
```

† For a change in value of a monitored data member to trigger rule evaluation, R++ syntax requires it to be first bound to a local variable (e.g., *spouse* is bound to sp).

The key difference between rules and ordinary C++ member functions is that changes to data members in the antecedent automatically cause the antecedent to be evaluated and, if it evaluates to TRUE, the consequent to be executed. Thus, rules are *data-driven*. In the example in Listing 2a, whenever a person's spouse changes, the `spouse_check` rule will automatically be evaluated. The key difference between R++ rules and rules in other rule-based languages is that R++ rules are *path-based*. This means that R++ rules can only reference data members which the class the rule belongs to has access to—typically through a pointer reference. This means that R++ rules *respect the object model*. The rules in Listing 2 are path-based because spouse and children are accessible via class `Person`. For those interested in a more in-depth understanding of R++, see the R++ home page.[6]

DESIGN PATTERNS

Building software systems is very difficult. Object oriented technology provides a way of modeling the world that often facilitates the analysis, design and implementation of such systems. Designing reusable object-oriented software is an even more difficult task—but a task that must be accomplished if dramatic increases in software productivity are to be achieved. *Design patterns* are reusable parts of software design or, more specifically, descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.[7] In this article, we are concerned with behavioral design patterns that characterize the ways in which classes or objects interact and distribute responsibility. In this section we will describe three behavioral design patterns used repeatedly throughout ANSWER. We will give the motivation and purpose for each pattern, example(s) of its use within ANSWER, how it was implemented in ANSWER using rules, how it would be implemented in C++, and the advantages of the rule-based implementation over the standard object-oriented implementation. This method of looking at rules in terms of design patterns allows us to describe and analyze our use of rules in a principled way, while still focusing on how rules were used in a real-world application.

Observer Pattern

An observer pattern defines a *one-to-many* dependency between objects so that when one subject object changes state, all its dependent observer objects are notified and updated automatically. The motivation for this pattern is that a side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects—but you do not want to achieve consistency by making the classes tightly coupled, because that reduces their reusability.[7] In the ANSWER project, the observer pattern is used for many things, including:

- *Memory management*: Each object *A* maintains a variable named *trash* and sets it to TRUE in its destructor; other objects which have dynamically allocated memory associated with *A* may have a trash rule which "observes" *A*'s *trash* data member and fires when trash is set to TRUE. This use of the trash rule can help eliminate memory leaks and dangling references.
- *Monitoring of communication links*: All objects that use a communication link "observe" the state of this link so that they can take appropriate action when the state changes (e.g., from the connected to the disconnected state).

Observer patterns are trivial to implement using R++ rules; all that is necessary is to have the observer objects contain a rule that "monitors" the value of the subject's data member of interest. In a language like C++, the observer pattern can be implemented using a *publish-subscribe* interaction. The subject *publishes* an interface for attaching and detaching observer objects. The observers then use this interface to *subscribe* and later, if desired, to unsubscribe. The observer object also needs to provide a callback function to the subject, to be invoked whenever a change occurs in the "monitored" data member. Once communication has been thus initialized, the subject can notify all observers whenever any change occurs to the monitored data members. Using this implementation of publish-subscribe, the subject need not know any specific details about each observer, but it *is aware* of them and how many observers there are. The rule-based implementation has the following advantages over the implementation with publish-subscribe:

- The subject object does not need to provide an interface method for attaching and detaching an observer object—R++ provides this implicitly.
- The subject object need not be aware of the observer objects at all. This contrasts with the publish-subscribe interaction, where the subject object needs to *maintain* a list of backpointers to the observer objects. This maintenance is error prone and not under full control of the subject object. If an observer gets deleted without explicitly unsubscribing first, then the subject is left with a *dangling reference* to the observer, which will likely lead to a core dump the next time the subjects observed data member changes. A similar problem can occur when the subject is deleted. Note that the rule-based implementation still requires backpointers to be maintained, but in this case all of the code is generated automatically by the R++ preprocessor—*not* by the programmer.

Notifier Pattern

A notifier pattern defines a *many-to-one* dependency between objects so that when a change in any of many objects occurs, the one object is notified. The motivation for this pattern is that it allows communication and logic to be centralized, which leads to more understandable and maintainable code. One example of this pattern in

ANSWER involves a "constraint violation" type rule. ANSWER tries to "auto-restore" failed 4ESS hardware components, a process that requires many steps and asynchronous communication with systems outside of ANSWER (including the 4ESS itself). While these steps are being executed, there are many conditions that must abort the autorestoral process, if they occur. A rule to implement this behavior is shown in Listing 3 (in the interest of space, six additional conditions have been omitted). Since the rule is data driven, the autorestoral process will be aborted automatically anytime any of the conditions become true.

Listing 3: Example of a notifier pattern

```
// aborts autorestoral of device, when appropriate
rule Device::auto_restore_abort {
   timer→expired                    ||  // step  timed-out?
   state→not_equal(OUT_SVC)         ||  // back in service?
   autorestore→disabled == TRUE     ||  // disabled from GUI?
⇒
   // abort and take appropriate action
};
```

To implement a many-to-one dependency like the one in Listing 3 without using rules is awkward. An accessor function is required for each data member in the antecedent of the rule and all changes to the data member must be through these accessors. Furthermore, each accessor must examine the change of the data member and either directly abort the autorestoral process, or call another method which evaluates all of the conditions together (this would be required if the antecedent were not a simple disjunction). The programmer is forced to manually implement a data-driven rule by distributing the triggers of the rule throughout the source code—R++, on the other hand, generates essentially the same code, but it does it automatically. Thus, the rule-based implementation requires much less programming effort and is more understandable and maintainable because all of the logic is centralized. Note that the advantages of the rule-based implementation still exist even if all the monitored conditions are located in the same object.

Client-Server Pattern

A client-server pattern defines a more complex communication pattern than either of the previous two patterns: There will generally be many clients and one or more servers and each client may need to be notified when a complex set of conditions is satisfied amongst one or more servers. There are many examples of the client-server pattern in ANSWER. One example involves the previously mentioned autorestoral process—the client is a device and the server implements the autorestoral process for that device. The advantages of a rule-based implementation are best shown via a hypothetical scenario, shown in Figure 1. The figure shows two servers (S1 and S2) and 4 clients (C1-C4). Each server has three publicly accessible data members, which represent information which may be of interest to the clients (e.g., variable *a* may indicate whether a communication link is up).

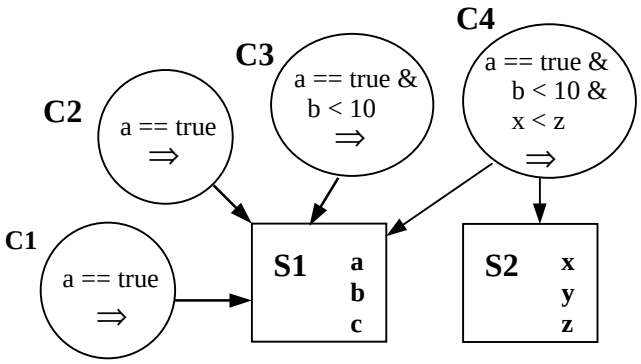

Figure 1: Client Server Pattern

The clients are interested in being informed when some situation has occurred, based on the values of the server's data members. In C++, the communication between the clients and servers would be established using publish-subscribe, as described earlier. Using this mechanism, if only C1 and C2 need to be considered, the best way to implement a solution is to send a notification to all clients when *a* becomes equal to *true*. In this case, the server has hardcoded into it the conditions of interest to the clients. If client C3 now needs to be supported, the programmer needs to either update the server to handle the new condition or employ a more flexible strategy which notifies each client when *any* variable changes and sends all of the current values (so the clients don't need to remember them). The later solution is superior since it does not require the server be constantly updated; however, it is inefficient and not very extensible (for client C4, both servers will need to broadcast all values on each change). An additional problem is that the programmer would need to implement the notion of relevant change—because client C1, for example, only wants to take action when *a* becomes *true* (e.g., a link initially comes up or goes down). The fundamental problem is that the best solution involves implementing the logic in the client, *where it belongs*, but C++ doesn't provide a simple way of doing this.

The solution to this problem using R++ is trivial; the client simply writes rules that monitors the variables in the server. R++ takes care of many of the details, including backpointer maintenance and the notion of relevant change. With R++, from the programmer's point of view, the client and server objects are now loosely coupled, with the conditional logic encapsulated within each client object. This leads to more understandable, maintainable, and reusable code.

CONCLUSION

The use of R++'s object-oriented rules significantly aided the design and implementation of ANSWER. Many of the advantages of using R++ were described in detail in this

article; rules facilitated communication between objects by implementing many low-level details via code generation and thus allowed us to program at a higher conceptual level, couple objects more loosely, and encapsulate responsibilities. Thus, R++ allowed us to produce more understandable, maintainable, and reusable code with fewer errors and less effort—3000 lines of R++ source code were translated by R++ into 17,000 lines of C++ code. In addition, we wrote 8,000 lines of C++ code ourselves, because some things are most naturally implemented using procedural code. This highlights another advantage of R++: It allows the programmer to implement a task using either rules or procedural code, whichever is the most appropriate for the given task. We believe that any project can benefit from these advantages and thus can benefit from R++.

In this article we focused on rules as a communication mechanism rather than as a way of declaratively encoding knowledge, even though we used rules in both ways (we did develop an expert system). We focused on rules as a communication mechanism to show that R++ can be helpful for general programming projects—not just for expert system applications. However, we should point out that object-oriented rules are extremely useful for implementing expert systems and we feel they are superior to traditional pattern-matching rules when a clear object model exists. R++ also has some advantages related to its relationship to C++—it is easy for C++ programmers to learn and it can (trivially) integrate with existing C++ code.

R++ is currently being used in several additional applications within AT&T and has being enhanced to support the ANSI based standard libraries. For more information, consult the R++ homepage.[6]

Acknowledgments

The authors would like to thank Jennifer Thien for supporting the use of R++ in ANSWER, Anoop Singhal for his contributions to the ANSWER expert system, and Anil Mishra for many discussions on the use of R++ within ANSWER.

References

1. Crawford. J. Dvorak, D., Litman, D. Mishra, A, and Patel-Schneider, P., "Device Representation and Reasoning with Affective Relationships", *Proceedings of the 14th International Joint Conference on Artificial Intelligence* (IJCAI-95), 1995.
2. Mishra, A., Ros, J., Singhal, A., Weiss, G., Litman, D., Patel-Schneider, P. Dvorak, D., Crawford, J., "R++: Using Rules in Object-Oriented Designs", *Addendum to Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA), October 1996.
3. Singhal, A. Weiss, G., and Ros, J., "A Model-Based Reasoning Approach to Network Monitoring", *ACM Workshop on Databases for Active and Real Time Systems* (DART-96), 1996.
4. Crawford, J., Dvorak, D., Litman, D., Mishra, A., and Patel-Schneider, P., "Path Based Rules in Object-Oriented Programming", *Proceedings of the 13th National Conference on Artificial Intelligence* (AAAI-96), August, 1996.
5. Litman, D., P. Patel-Schneider, and A. Mishra, "Modeling Dynamic Collections of Interdependent Objects Using Path-based Rules", *Proceedings of the 12th Annual Conference on Object Oriented Programming, Systems, Languages, and Applications* (OOPSLA 97), 1997.
6. R++ home page: http://www.research.att.com/sw/tools/r++.
7. Gamma, E. Helm, R., Johnson, R. Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.