18

Extensible and Reusable Role-Based Object-Oriented Security

S. A. Demurjian, T. C. Ting, M. Price[a], and M.-Y. Hu[b]

[a]*CS&E Dept., University of Connecticut, Storrs, CT 06269, USA*
[b]*IBM Corp., 1131 Mamaroneck Ave., White Plains, NY 10605, USA*
e-mail: {steve,ting}@eng2.uconn.edu, myhu@vnet.ibm.com

Abstract

User-role based security (URBS) has drawn significant attention in recent years for its ability to customize security privileges according to the responsibilities of individual user roles. In object-oriented applications, the public interface of each class contains methods for all potential users of the class. URBS can be introduced to promote a strategy that controls access on a role-by-role basis, with different roles having access to specific subsets of each public interface based on their responsibilities within the application. This paper continues these efforts by investigating approaches for extensible and reusable URBS enforcement mechanisms for object-oriented systems. Such approaches should insulate software engineers from security concerns while simultaneously embedding the URBS policies into compiled applications that then behave differently based on an individual's role. We consider generic security classes that stress uniformity, encapsulate security details, and promote software reuse. We explore exception handling as an vehicle for achieving dynamic role-based behavior. Together, generics and exception handling yield an approach that attains software reuse and software evolution.

Keywords

User-role based security, object-oriented systems, generics, exception handling.

1. INTRODUCTION

Object-oriented design and implementation techniques rely on the public interface of a object type/class to collect all of the permissible operations/methods needed by all potential users. Thus, methods placed in the public interface are available to all potential users regardless of their intended responsibilities within

P. Samarati et al. (eds.), *Database Security*

the object-oriented system/application, i.e., there is no way to prevent access by a user to a method in the public interface. User-role based security (URBS) approaches, a growing interest in recent years [12, 17, 19], are promoted for their ability to customize access based on a role's responsibilities. Combining object-oriented public interface concepts with URBS leads to a strategy that controls access to the public interface on a role-by-role basis, where different user roles would have access to specific and limited subsets of public interfaces based on their overall responsibilities. Such a scheme finely tunes the security policy and is an important first step to minimizing misuse and corruption.

Over the past five years we have focused on URBS as outlined in the previous paragraph by extending the ADAM environment [2, 9, 10, 11]. ADAM, short for Active Design and Analyses Modeling, is a language-independent, object-oriented design environment [7] that can generate compilable code in C++ (GNU C++ and Ontos C++ - an object-oriented database system), Ada1983, Ada95 [6], and Eiffel [13]. In our approach, user roles are defined in a hierarchy. Once a role has been defined, privileges can be established by assigning methods (positive - can invoke a method) and prohibiting methods (negative - can't invoke a method or a method that calls the prohibited method, and so on).

In our more recent effort on URBS [4], two approaches for extensible and reusable C++ URBS mechanisms were proposed that try to insulate the software engineer from security considerations while simultaneously insuring that the object-oriented system enforces the required URBS. The result would have the URBS policies embedded into the executable image of an application, thereby insuring that the roles of users dictate the application's functionality at runtime. A follow-up to that effort raised and discussed two crucial issues related to attaining URBS for object-oriented systems [5], namely:

- Generic security classes that stress uniformity and promote software reuse.
- Exception handling for dynamic role-based behavior.

This paper explores these two issues in greater detail by presenting candidate approaches that enforce URBS in object-oriented/C++ applications. One approach that utilizes generics has both an actual solution and a desired solution. The difference is that the desired solution doesn't compile due to C++ language limitations. Another approach for exception handling also utilizes generics to provide solutions that are easily incorporated in all object types/classes that require URBS capabilities. For both approaches, the key is to insure the attainment of the object-oriented principles of *software reuse*, *software evolution*, and *extensibility* [3].

Our early efforts in URBS have been strongly influenced by [15, 18]. Our recent work involving enforcement mechanisms has been impacted by work in object-oriented programming/design related areas. There has been work on *aspects* as a mechanism for object-oriented data models to extend a given class with new capabilities, including, roles [16]. In this work, new operations and data are possible for new roles; in our work, the class has a superset of operations/data to which the roles require selected access. Another effort allows different classes in an application

to have different subjective views [8], which is similar to our approach of having different methods of the public interface available to different users based on their roles. More recent work has focused on composing these subjective views with only scant mention of implementation support in C++ [14]. Finally, an effort on role-based access control for object technology [1] takes a very similar approach to our own efforts. When different roles require specific access to a class, subclasses for the roles are created to turn-on/turn-off the appropriate access.

The main purpose of this paper is to report on our ongoing efforts that are exploring the use of generics and exception handling for URBS in object-oriented systems. In Section 2, we provide background material on: ADAM and its URBS capabilities, and the previous approaches to enforcement mechanisms [4]. Section 3 details the direct updates of the work presented in [4] by relating experiences on incorporating URBS into object-oriented systems and discussing the utilization of generics. Section 4 contains an in-depth examination of new work that demonstrates how exception handling can be utilized to realize URBS enforcement. Finally, Section 5 summarizes the paper and indicates future work.

2. BACKGROUND

2.1. ADAM and URBS

The ADAM environment stresses language independence by focusing on design and allowing code to be generated in a variety of target languages. Designs are entered using profiles [7, 9], which require software engineers to supply detailed requirements on the semantic content and context for all constructs of the application, and also provide on-demand and automatic analyses for feedback when conflicts/inconsistencies are detected. To segment the design process into logical parts, ADAM has design phases. In the object-type-specification phase of ADAM, the designer can define the object types (OTs), their attributes, and methods through associated profiles [7]. A subset of ten OTs for a health care application (HCA), based on prior work [9], is shown in Figure 1. `Visit`, `Prescription`, and `Test` are subtypes of `Item`, for different medical procedures for a patient. Each OT in ADAM is identified, for instantiation purposes, as being an abstract class (shown (A) in Figure 1), a regular class (shown (R), signifying instantiation anywhere in the hierarchy), or a database class (not shown (D), signifying instantiation at only leaf nodes). Additionally, each subtype inherits behavior (data and methods) from the parent in many ways; *Public* signifies that only public methods are passed, while *Full* indicates that all private and public methods and data are inherited. In the second phase of ADAM (not shown), the designer can define the relationship types between different OTs.

A third phase of ADAM supports URBS through a user-role definition hierarchy (URDH). The URDH characterizes the different kinds of individuals (and groups) who all require different levels of access to an application. Figure 2 shows a partial URDH for HCA. At the lowest level in the figure, user roles (URs) allow the security software engineer to assign particular privileges for individual responsibilities, e.g., `Staff_RN` and `Education` are URs. To represent common responsibilities among

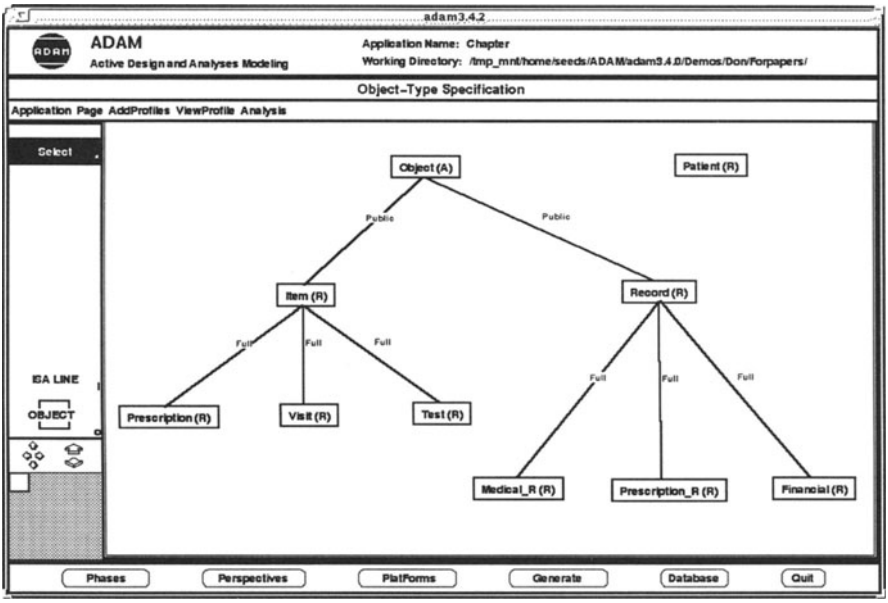

Figure 1 Object types of a health care application (HCA).

URs, a user type (UT) can be defined, e.g., `Nurse` and `Physician`. UTs can be grouped in to one or more user classes (UCs), e.g., `Medical_Staff`. Privileges that are supplied to a UT(UC) are passed on to its URs (UTs and their URs).

To characterize the capabilities of UCs, UTs, and URs, in the URDH, a node profile, is utilized [11]. A node profile contains: a name for the node (UR, UT, or UC) and a prose description of its responsibility; a set of assigned methods (positive privileges - methods of the public interface that the UR/UT/UC can invoke); a set of prohibited methods (negative privileges - methods of the public interface that the UR/UT/UC can't invoke); and a set of criteria for relating URDH nodes. A security designer assigns relevant methods from the HCA to realize the positive accesses for `Staff_RN`. This is augmented with methods like `Set_Prescription` that `Staff_RN` can't use. Consistency criteria can establish equivalence and subsumption associations between different URDH nodes based on privileges, e.g., `Staff_RN` subsumed by `Manager`. Finally, in a fourth phase of ADAM, the URDH can be utilized to grant/revoke URs to individuals to form an authorization list.

2.2. URSA and UCLA

In this section, two approaches to generating code for a URBS enforcement mechanism [4] are briefly reviewed using the HCA example from the previous section. These approaches function under the assumption that users utilize tools that embody the apropos security code to enforce the required URBS policy. The user-role-subclassing approach (URSA) utilizes inheritance to derive new user-role subclasses for each OT in an application. In URSA, each subclass of an OT contains the assigned and prohibited methods by the UR on that OT. In URSA, each class

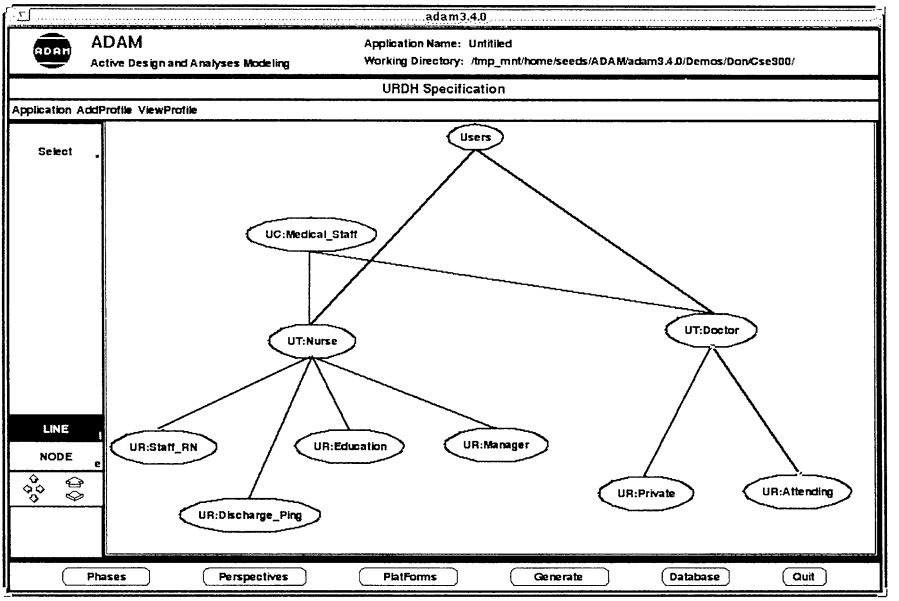

FIgure 2 The URDH of the HCA.

of the application has a group of subclasses, based on the different roles that have some subset of assigned and/or prohibited methods from the class. As subclasses, the basic concept is to inherit the methods that are assigned in a positive sense (can invoke and will return results) while simultaneous inheriting the prohibited methods in a negative sense (can invoke but wouldn't return results). The key for URSA is to turn-off prohibited methods. To achieve this result, the C++ concepts for virtual functions and void return types are employed:

```
class Prescription {
  public:
    virtual Get_Prescription_No(...); virtual Set_Prescription_No(...);
    virtual Get_Pharmacist_Name(...); virtual Set_Pharmacist_Name(...);
    virtual Get_Medication(...);      virtual Set_Medication(...);
  private:
    char* prescription_no, pharmacist_name, ...   };

class Staff_RN_Prescription : public Prescription {
  public:
    virtual void Set_Prescription_No(...) {return;} // Turned-Off
    virtual void Set_Pharmacist_Name(...) {return;} // Turned-Off
    virtual void Set_Medication(...) {return;}   }; // Turned-Off

class Attending_MD_Prescription : public Prescription {
  public:
    virtual void Set_Pharmacist_Name(...) {return;} }; // Turned-Off
```

If the Set_Prescription_No method is requested by a individual whose role is Staff_RN, then the method associated with Set_Prescription_No of the subclass Staff_RN_Prescription is executed and no value is returned. If the access was made by an Attending_MD role, then the actual method would be executed to allow the prescription to be set. This is accomplished by the concepts of substitutability, dispatching, and mutability in the runtime environment. Extensibility is supported, in part, since if new roles are needed, subclasses of Prescription can be added. The problem with URSA is that the software engineer must write application code that utilizes either if or switch statements which are dependent on roles. If existing roles are modified or the classes are changed, the application must be first changed and then recompiled. Note also that URSA is very similar in concept to the approach utilizing role classes. [1].

The URDH-class-library approach (UCLA) employs inheritance to implement the enforcement mechanism from the combined perspective of the application's URDH and OT/class library. In UCLA, a new class hierarchy is provided, where each class represents a URDH node and the access rights of the node are specified via a set of validation methods. For each URDH node, positive method access is defined based on the assigned methods that have been specified. As the application executes, each method invocation must validate against the current UR. For the same example given earlier, the class hierarchy would be:

```
URDH:  UC: Medical_Staff --- UT: Nurse --- UR: Staff_RN
       Assigned methods:
         Get_Prescription_No(), Get_Pharmacist_Name(), Get_Medication()

New URDH Class Library:
   class Root: All Check Methods defined to return False;
   class Users: public Root
   class Medical_Staff: public Root
   class Nurse: public Users, public Medical_Staff {
         int Check_Prescription_Get_Medication() {return True;} };
   class Staff_RN: public Nurse {
         int Check_Prescription_Get_Prescription_No() {return True;}
         int Check_Prescription_Get_Pharmacist_Name() {return True;} };
```

Note that only a subset of the URDH nodes from Figure 2 are given above. The Root class includes new Check methods which are defined for all application methods from all classes to return False. These Check methods will be turned on at lower levels (UC/UT/UR) by the assigned methods of the URDH. These Check methods are also utilized to change the code generated for each class:

```
class Prescription: public Item {
 public:
  Prescription(Root* u,char* Name,char* D,int No,char* Name1,char* Med);
  int Get_Prescription_No(); void Set_Prescription_No(int No); ... ; }

int Prescription::Get_Prescription_No()
   { if(current_user->Check_Prescription_Get_Prescription_No())
```

```
        return(Prescription_No);
      else return(NULL);  }

void Prescription::Set_Prescription_No(int No)
   { if(current_user->Check_Prescription_Set_Prescription_No())
       Prescription_No=No;  }
```

At runtime, a user's role enables the current_user pointer to invoke the apropos Check method, customizing the results of the invocation. The UCLA approach resolves undo problems, since no changes are made to actual attributes by unauthorized roles, there are no actions to be undone. As URs are added, only the URDH class library must be recompiled. Similarly, if a UR's assigned and/or prohibited methods change, only the URDH classes would need to be modified and recreated. Changes to the application still impact on both class libraries.

3. GENERIC SECURITY CLASSES

Generic security classes for URBS (via templates in C++) would streamline the process that automatically includes URBS code in an application. Generic security classes would acquire the domain specifics (user roles, assigned methods, prohibited methods, etc.) and then provide the necessary authentication and enforcement. By encapsulating URBS concepts into generic classes we can achieve uniformity as an instrumental step towards the attainment of software reuse. Generic security classes could be designed, implemented, tested, and validated, independent of an application, providing a higher degree of assurance to security conscious users. This section reviews experiences in a semester project in utilizing URSA and UCLA in Section 2.2 for a more realistic application. Then, an approach to using generics for URSA developed by one of the co-authors of this paper is detailed. A corresponding solution for UCLA was also developed, but has been omitted, since it is strongly similar in concepts and details.

3.1. Experiences from graduate project

During the Fall 1995 semester, a semester project for a graduate course at The University of Connecticut involved applying the URSA and UCLA approaches presented in Section 2.2 to a more realistic example. In the project, a class library with 1700 lines of code was modified to support URBS for the URSA and UCLA approaches. There were 8 classes in the library with 120 total public methods. The comments of the twelve students doing the project indicated that the majority of the effort for both approaches was busy work; in URSA with generating the new subclasses, and in UCLA with generating the class library and check methods for the URDH. One student implemented a realistic URDH that required 1600 check methods in the UCLA approach. The unanimous recommendation was that automatic code generation by ADAM was critical to reduce the busy work and minimize errors. Additionally, the students made similar conclusions to the ones given in Section 2.2 regarding the advantages and drawbacks of URSA and UCLA.

As presented, both URSA and UCLA have limitations that inhibit their usage in an environment where the security policy is constantly evolving. This was also

the observation of the students from the graduate project; when changes to the security policy were made (changes to assigned and/or prohibited methods for one or more URs), these changes were invasive in the code for both URSA and UCLA. The students also noted that the addition of a new UR was easy to accommodate in UCLA and required only changes to the URDH class library. Attempts to utilize generics for URSA and UCLA in the project met with only minimal success, and were limited to function templates.

3.2. The generic user-role subclassing approach

The generic user-role subclassing approach (GURSA) reformulates the techniques given in Section 2.2 by introducing a generic prescription class that encapsulates user-role specific access. The `Generic_Prescription` template class is parameterized with a single value, the user-role type (`URType`), which represents the UR of the individual that is attempting to access the `Prescription` class.

```
#include "s_pres.h"  // Staff_RN_Prescription Class from Section 2.2
#include "a_pres.h"  // Attending_MD_Prescription Class from Section 2.2

template <class URType> class Generic_Prescription {
private:
  URType   *current_UR;
public:
        Generic_Prescription(){current_UR=new URType("", "", 0, "", "");}
  int   Get_Prescription_No(){return(current_UR->Get_Prescription_No());}
  void  Set_Prescription_No(int No){current_UR->Set_Prescription_No(No);}
  char* Get_Pharmacist_Name(){return(current_UR->Get_Pharmacist_Name());}
  void  Set_Pharmacist_Name(char* Name)
                            {current_UR->Set_Pharmacist_Name(Name);}
  char* Get_Medication() {return(current_UR->Get_Medication());}
  void  Set_Medication(char* Med) {current_UR->Set_Medication(Med);}
  void  copy_object(Prescription *P) {current_UR->copy_object(P);}
}; // End of Generic_Prescription template class header
```

When a variable of type `Staff_RN_Prescription` or `Attending_MD_Prescription` is created, the UR is passed in as a parameter. This has the effect of allowing the `current_UR` pointer to correctly choose the methods associated with each of these classes at runtime based on the UR of the individual attempting to access a `Prescription` instance. Recall from Section 2.2 that each of these two classes had methods redefined that turned-off prohibited methods.

Given the template class, there are two implementation options that we have examined. The first, which compiled and executed, is as follows:

```
#include "item.h"           // Item from Figure 1.
#include "pres.h"           // Prescription from Figure 1.
#include "s_pres.h"         // Staff_RN user role subclass.
#include "a_pres.h"         // Attending_MD user role subclass.
#include "generic_pres.C"   // Generic_Prescription template class.

main()
```

```
{  Prescription* P;    int Number;
   char user_name[64], char user_role[64], *Medication, *Name;

   cout << "Please input your name:"; cin >> user_name;
   cout << "Please input your role:"; cin >> user_role;

   P = new Prescription("Jessica", "3-9-95", 1, "Kitty", "Cold");

   if (strcmp(user_role, "Staff_RN")==0)
     { Generic_Prescription<Staff_RN_Prescription> *SP
         = new Generic_Prescription<Staff_RN_Prescription>();
       SP->copy_object(P);  //copy Attributes from the parent
       SP->Set_Prescription_No(200); // Fails
       Number=SP->Get_Prescription_No(); // Succeeds
       cout << "Number==" << Number << "\n";
     }
   else
     if (strcmp(user_role, "Attending_MD")==0)
       { Generic_Prescription<Attending_MD_Prescription> *AP
           = new Generic_Prescription<Attending_MD_Prescription>();
         AP->copy_object(P);  //copy attributes from the parent
         AP->Set_Prescription_No(200); // Succeeds
         AP->Set_Pharmacist_Name("Steve"); // Fails
         Number=AP->Get_Prescription_No(); // Succeeds
         Name = AP->Get_Pharmacist_Name(); // Succeeds
         cout << "Number==" << Number << "\n";
         cout << "Name==  " << Name << "\n";
       }
} // End of main
```

The advantage to the template `Generic_Prescription` is that the current user role is encapsulated and hidden from the software engineer who writes the above code. However, the stronger disadvantage is the presence of the conditional statement in the code. Such conditionals will be present in any application method that needs to control access to information based on user role. Conditionals severely limits extensibility, since they are pervasive throughout the application code. Conditionals also require the software engineer to have a significant knowledge of URBS requirements for the application, to correctly insert conditionals into the code.

These disadvantages led us to explore a second implementation alternative that replaces the conditional with:

```
if (strcmp(user_role, "Staff_RN")==0)
    Generic_Prescription<Staff_RN_Prescription> *PP
        = new Generic_Prescription<Staff_RN_Prescription>();
else
   if (strcmp(user_role, "Attending_MD")==0)
       Generic_Prescription<Attending_MD_Prescription> *PP
          = new Generic_Prescription<Attending_MD_Prescription>();
```

```
// Would allow one block of user-role independent code.
PP->copy_object(P);  //copy attributes from the parent
PP->Set_Prescription_No(200);      // Success or Failure Based on UR
PP->Set_Pharmacist_Name("Steve"); // Success or Failure Based on UR
Number=PP->Get_Prescription_No(); // Success or Failure Based on UR
Name = PP->Get_Pharmacist_Name(); // Success or Failure Based on UR
cout << "Number==" << Number << "\n";
cout << "Name==  " << Name << "\n";
```

In this situation, once the UR has been identified, the single pointer PP (which replaces SP and AP) is created and initialized. Then, all subsequent code will behave differently based on PP's initialization. The key concept is that the conditional occurs one time when a user first begins executing an application or tool. After this has occurred, subsequent access via a common pointer will be dynamically determined in a fashion that is consistent with the initialization of PP. The problem is that this code won't compile. This is since C++ won't allow the same variable to be declared with different types in the same scope. If it would, during runtime, a decision based on dynamic information, in this case, the UR, would call the correct methods. In the previous example, SP and AP were in different scopes; in this case, the attempt is to place PP in one scope with its type decided at runtime. The above would be an elegant and practical alternative if the language, compiler, and runtime support existed for the needed variable and type declarations.

4. EXCEPTION HANDLING FOR URBS

Exceptions are traditionally intended to handle situations where an error in an executing program can be anticipated, caught, and processed (e.g., divide by zero). Exceptions are attractive from a security perspective since they concentrate error-handling code in one location. Thus, there is a chance that URBS enforcement code be placed in exception handlers and hidden from software engineers. Also, a UR trying to access a method that was prohibited from use is similar in concept to an error occurring. In this context, it is possible to raise exceptions when a UR attempts to access a method that was not allowed. A raised exception can be handled in a number of ways. First, it might simply be that when a UR has a prohibited method, an exception is raised, and handled by having the method not execute. Second, the raised exception might be handled by alerting apropos individuals to the potential security violation.

This section explores two approaches that use exception handling as supported by C++ to enforce URBS. The *basic exception approach (BEA)* utilizes straightforward techniques to embed exception handling code directly into each class. The *generic exception approach (GEA)* incorporates concepts of template classes that results in a significant core of generic code to encapsulate and hide exception handling details. In both approaches, when a method is invoked, the UR of the current user is checked to verify if access can be granted. If not, an exception is raised, and processed accordingly. We also explore advanced exception handling techniques to

illustrate where more complex code can be placed in the event of a security violation. Finally, we provide a discussion of code generation for exceptions as related to the ADAM environment.

4.1. A basic exception approach

Exception handling in C++ involves a number of programming language concepts and constructs. The `try` construct is utilized to encapsulate a block of code (in our case, a method call) that has the potential to raise an exception. As the code within the `try` block is executing, various conditions can be checked, and when the correct situation occurs (in our case, an attempt to access by an unauthorized UR), an exception can be raised using the `throw` construct. This thrown exception is then processed by a `catch` block that typically follows the original `try` block. In our case, the `catch` block will be used to process the security violation.

In the basic exception approach (BEA), each class is modified to include a set of methods for exception handling. This is illustrated in the header file for the `Prescription` class. Note that various details such as include files and variable definitions have been omitted to both simplify and clarify the presentation.

```
class Prescription: public Item {
  // Private data has been omitted
public:
  Prescription(char* Name,char* D,int No,char* Name1,char* Med);
  int   Get_Prescription_No();          void  Set_Prescription_No(int);
  char* Get_Pharmacist_Name();          void  Set_Pharmacist_Name(char*);
  char* Get_Medication();               void  Set_Medication(char*);
  int   rtn_int_check_valid_UR(int);
  char* rtn_str_check_valid_UR(char*);
  void  set_int_check_valid_UR(int*,int);
  void  set_str_check_valid_UR(char*,char*);
  void  Check_UR(); // Method to check if UR can access method.
  class Unauthorized_UR { }; // Exception handling methods follow.
}; // End of Prescription class header
```

There are six new methods that have been added. `Check_UR` is needed to verify that the current UR can invoke the desired method. For the purposes of this first example, we'll assume a simple table lookup. The class `Unauthorized_UR` is an exception handling class. While the functionality in this case is null, it can be totally expanded to handle complex situations based on the exception that has been raised. An example of this is given in Section 4.3. The remaining four methods are used to handle exceptions for the Prescription methods that return (`rtn`) information, and `set` values.

To clearly understand how all of the pieces interact, a portion of the implementation file for the `Prescription` class is provided.

```
int Prescription::Get_Prescription_No()
{
  return(rtn_int_check_valid_UR(Prescription_No));
}
```

```
int  Prescription::rtn_int_check_valid_UR(int AInt)
{
  try {// try block has potential to raise exception
     this->Check_UR();
  }
  // catch block processes any raised exceptions
  catch (Prescription::Unauthorized_UR) {
    cout << "Attempt to access by unauthorized UR" << endl;
    return(INT_NULL);
  }
  return(AInt);
}

void Prescription::Check_UR()
{ // Pseudo-code to simply illustrate concepts!!
  // Shows invalid URs for Get_Prescription_No method.
  if ((Current_User->Get_User_Role() != Staff_RN)
   || (Current_User->Get_User_Role() != Attending_MD))
     throw Unauthorized_UR(); // throw raises exception
}
```

The method Get_Prescription_No returns the result of the rtn method which is passed the number. The rtn method begins with the try block to verify whether the role of the current user has permission to invoke the Get_Prescription_No method. If not, the exception Unauthorized_UR is thrown by Check_UR. If the exception was thrown, it is then captured by the catch block in the rtn method. In the example, the catch block prints an error message and returns a null integer. If the exception was not thrown, control drops through to return the requested integer, in this case, Prescription_No.

The main program that would work with the Prescription class and its exception handler is as follows:

```
User* Current_User;
main()
{  Prescription* P;    char* Name, Date, Medication, name, role;
   int Number;         char user_name[64], user_role[64];

   cout << "Please input your name:"; cin >> user_name;
   cout << "Please input your role:"; cin >> user_role;

   Current_User = new User(user_name, user_role);
   name = Current_User->Get_User_Name() ;
   role = Current_User->Get_User_Role() ;

   // Data is set as: MDName, Date, Presc#, PharName, Medication
   P = new Prescription("Lois", "2-13-96", 1, "John", "Aspirin");
   Name = P->Get_Physician_Name();
```

```
    cout << "Name==" << Name << "\n";
    Date = P->Get_Date();
    cout << "Date==" << Date << "\n";
    Number = P->Get_Prescription_No(); // Will fail for UR other than
                                       // URs Staff_RN or Attending_MD
    cout << "Pre_No==" << Number << "\n"; // If fails - Null
    P->Set_Pharmacist_Name("MeiYu");  // Will fail for Staff_RN
    Name = P->Get_Pharmacist_Name();  // Unchanged if Staff_RN
    cout << "Name==" << Name << "\n";
} // End of main
```

Depending on the privileges that were established in the URDH, the results from the Get and Set methods above will be dictated by the user role of Current_User. All of the information regarding the URDH, assigned, and prohibited methods is included within the Prescription class (not shown) and is available via the Check_UR method as was previously discussed.

There are two advantages to BEA. First, the code for the rtn, set, and Check_UR encapsulates the URBS code, hiding these details from the software engineer. Second, once the user role has been established (via Current_User), the rest of the code given in main() doesn't require the identification of UR. This is in contrast to the GURSA code given in Section 3.2, where SP and AP were specific user-role pointers. However, there are a number of obvious disadvantages. First, the exception handling code is pervasive throughout Prescription; extensibility is definitely hindered if changes are needed to the class. Second, there is a great deal of replicated code, particularly in the set and rtn methods. Thus, reuse is not attained.

4.2. A generic exception approach

The generic exception approach, GEA, is intended to alleviate the disadvantages of BEA while still maintaining its strengths. This is accomplished by the development of a template class that generalizes the exception handling code. This template class can then be reused throughout the system for all application classes that require URBS enforcement. At the core of GEA is the template class Gen_Security.

```
template <class Type> class Gen_Security {
  protected:
    int  C_ID; // Class ID
    int  M_ID; // Method ID
  public:
         Gen_Security() {C_ID = M_ID = 0;}
    void  GS_Check_UR();
    class Unauthorized_UR { }; // Exception
    int   rtn_int_check_valid_UR(int);
    float rtn_flt_check_valid_UR(float);
    char* rtn_str_check_valid_UR(char*);
    char  rtn_chr_check_valid_UR(char);
    void  set_int_check_valid_UR(int*,int);
    void  set_flt_check_valid_UR(float*,float);
```

```
    void  set_str_check_valid_UR(char*,char*);
    void  set_chr_check_valid_UR(char,char);
    void  prt_int_check_valid_UR(int);
    void  prt_flt_check_valid_UR(float);
    void  prt_str_check_valid_UR(char*);
    void  prt_chr_check_valid_UR(char);
}; // End of Gen_Security template class header
```

Notice that this class header contains rtn and set methods for the four primary data types, along with the GS_Check_UR and Unauthorized_UR methods that are consistent with BEA in Section 4.1. In addition, print methods (prt prefix) have been provided for the same data types. Note also that two crucial pieces of protected data are maintained, namely, the class and method identifiers. C_ID is unique across an application, while M_ID is unique within a class. Together, they yield a unique identifier for every method. Finally, Type is the class name of the class that requires URBS enforcement.

A portion of the code for the methods of the template Gen_Security is provided below. Like the BEA example, repetitive code is omitted for brevity. Only one of the prt methods is shown; rtns are similar to BEA with generic syntax overlaid, while sets simply have an assignment in place of the output statement in prt.

```
template<class Type> void Gen_Security<Type>::GS_Check_UR()
{ // Call Check of Owner Class
 if (Type::Check_UR(C_ID+M_ID) == FAIL) throw Unauthorized_UR();
}

template<class Type> void
    Gen_Security<Type>::prt_str_check_valid_UR(char* AString)
{
  try {// try block has potential to raise exception
    this->GS_Check_UR();
  }
  // catch block processes raised exception
  catch (Gen_Security::Unauthorized_UR) {
    cout << "Attempt to access by unauthorized UR" << endl;
    return; // return control without printing
  }
  cout << AString << endl; // output if no exception raised
}
```

The most interesting aspect of this code involves the GS_Check_UR method. At compile and then runtime, the if statement is parameterized appropriately. Thus, if Prescription is the class to be controlled, the if statement would be:

```
if (Prescription::Check_UR(C_ID+M_ID) == FAIL) throw Unauthorized_UR();
```

This insures that the correct Check_UR method for a Prescription instance is invoked. Like BEA, the effect of catch in the prt method can be changed to be more relevant in the event of an attempted access by an unauthorized user role.

For example, automatic notification of a relevant security individual can be made. In addition, conditions can be set that allow more complex actions to occur when an unauthorized UR access has been detected.

To illustrate the usage of Gen_Security, the header files for the Item and Prescription classes are given. Notice that an include file is used to bring in the Gen_Security template.

```
#include "gen_security.C"
class Item {
  // Private data has been omitted
public :
  Item(char* Name,char* D);
  char* Get_Physician_Name();
  void  Set_Physician_Name(char*);
  void  Print_Physician_Name();
  char* Get_Date();
  void  Set_Date(char*);
  void  Print_Date();
  static int   Check_UR(int); // Class specific URBS
  static int   Assigned_Methods(int); // Assigned methods
  Gen_Security<Item> Item_Sec; // Template Declaration
}; // End of Item class header

class Prescription: public Item {
  // Private data has been omitted
public:
  Prescription(char* Name,char* D,int No,char* Name1,char* Med);
  int   Get_Prescription_No();
  void  Set_Prescription_No(int);
  void  Print_Prescription_No();
  char* Get_Pharmacist_Name();
  void  Set_Pharmacist_Name(char*);
  void  Print_Pharmacist_Name();
  char* Get_Medication();
  void  Set_Medication(char*);
  void  Print_Medication();
  static int  Check_UR(int); // Class specific URBS
  static int  Assigned_Methods(int); // Assigned methods
  Gen_Security<Prescription> Pres_Sec; // Template declaration
}; // End of Prescription class header
```

Unlike BEA, only two methods and one variable declaration have been added to each class. The template declaration is parameterized with either Item or Prescription via the corresponding variable declaration of Item_Sec or Pres_Sec.

The usage of these three methods and the interactions with the Gen_Security template are illustrated in the subset of the implementation code for Prescription:

```
void Prescription::Print_Pharmacist_Name()
{
```

```
  Pres_Sec.prt_str_check_valid_UR(Pharmacist_Name);
}

int Item::Check_UR(int unique_method_id)
{
  return(Item::Assigned_Methods(unique_method_id));
}

int Prescription::Assigned_Methods(int meth_id)
{ // For now - simulate by hard-coding response.
 if (  (strcmp(Current_User->Get_User_Role(), "Staff_RN") == 0)
    || (strcmp(Current_User->Get_User_Role(), "Attending_MD") == 0))
     return SUCC;
 else
     return FAIL;
}
```

The only difference between BEA and GEA is that the call to a `prt` (or `rtn` or `set`) method must be prefaced with the variable `Pres_Sec`. Note also that by having the unique method identifier passed in as a parameter, it will be possible to implement `Assigned_Methods` as a runtime data structure (linked list) to verify whether the current UR has been given access to the method `meth_id`. This information can be loaded into the runtime data structure from a file. When privileges are changed, then the file is updated and the compiled code still works correctly. Finally, the `main()` for GEA has been omitted since it is identical to the main for BEA in Section 4.1.

GEA enhances the advantages as BEA by encapsulating all exception handling for a class into a template. This is turn promotes software reuse, since all classes that require URBS can utilize the `Gen_Security` template, as demonstrated with `Item` and `Prescription`. While code for methods defined on `Prescription` must be changed (see `Print_Pharmacist_Name`), like BEA, the changes don't alter the signature of the methods. From the perspective of the software engineer writing code, the URBS enforcement via exceptions is hidden.

4.3. Advanced exception handling

The exception handling techniques discussed in Sections 4.1 and 4.2 provide a starting-off point for more advanced capabilities. Specifically, there can be multiple exceptions in the `Gen_Security` template, where each exception contains a body of code for processing security violations. This is illustrated below with the reformulated `Gen_Security` template:

```
template <class Type> class Gen_Security {
  protected:
    int  C_ID; // Object ID
    int  M_ID; // Method ID
  public:
    Gen_Security() {C_ID = M_ID = 0;}
    void  GS_Check_UR();
```

```
    class Unauthorized_UR {
      public:
        int  meth_id;
        Unauthorized_UR(int m_id) {
          meth_id = m_id;
          // Remaining code to process exception.
          cout << "Attempt to Access Method --> " << meth_id << endl;
          cout << "by Unauthorized User Role!!!!" << endl;
        }
    }; // End Unauthorized_UR exception

    class Logauthorized_UR {
      public:
        int  meth_id;
        FILE *f_id;
        Logauthorized_UR(int m_id) {
          meth_id = m_id;
          // Remaining code to process exception.
          // Print message to FILE f_id that logs all
          // accesses to methods by authorized URs.
        }
    }; // End Logauthorized_UR exception

    // rtn, set, prt methods as previously given without changes
}; // End of Gen_Security template class header

template<class Type> void Gen_Security<Type>::GS_Check_UR()
{ // Call Check of Owner Class
  if (Type::Check_UR(C_ID+M_ID) == FAIL)
    throw Unauthorized_UR(C_ID+M_ID);
  else
    throw Logauthorized_UR(C_ID+M_ID);
}
```

Each exception can have public and/or private data, and be defined to perform any task needed to reconcile the raised exception. Notice also that a second exception has been added that is intended to log all authorized accesses to methods by URs. The presence of another exception necessitates additional catch blocks in all of the Gen_Security method implementations.

```
// Two catches replace one in all relevant Gen_Security methods
catch (Gen_Security::Unauthorized_UR) {
  cout << "Attempt to access by unauthorized UR" << endl;
  return; // return control without printing
}
catch (Gen_Security::Logauthorized_UR) {
  cout << "Logging of authorized access by UR" << endl;
}
```

These catch blocks are checked sequentially, with either control returning to the invoking method (`Unauthorized_UR` catch) or dropping through to continue execution (`Logauthorized_UR` catch).

4.4. Code generation and ADAM

Our approach to supporting URBS via exception handling is consistent with the code generation capabilities provided by the ADAM environment. Object types in ADAM are defined to have both methods and attributes. For each attribute, the software engineer is provided with an option that causes ADAM to automatically generate get, set, and print methods. Thus, the `Gen_Security` template given in Section 4.2 is consistent with ADAM's code generation capabilities since it provides for `rtn`, `set`, and `prt` methods for all basic data types. The code presented in both Sections 4.2 and 4.3 is also very consistent and regular, lending itself to automatic code generation by ADAM. In fact, ADAM already generates the non-security components of the header files for `Item` and `Prescription` as presented in Section 4.2. It is a trivial task to generate needed include files and method declarations for security related actions. Changes to the ''`.c`'' files can also be made in a similar fashion. We expect that a C++ code generator for exception handling in ADAM using the approach in Section 4.2 to be available shortly.

5. CONCLUDING REMARKS AND FUTURE WORK

This paper has reported on our continuing efforts to support user-role based security for object-oriented systems and applications. Our previous efforts have focused on two inheritance-based approaches to support URBS, URSA and UCLA, as reported in [4] and briefly reviewed in Section 2.2. The drawbacks of both approaches, particularly in the areas of software reuse and the need for software engineers to understand security details, have spurred us to investigate other alternative approaches. Section 3.2 examined one approach that utilized a C++ template class in an attempt to fashion a generic URSA. Unfortunately, compiler and language limitations still required the use of conditional statements and user-role specific code, which greatly reduces the ability to extend and evolve an object-oriented application. In searching for alternative approaches, exception handling capabilities were examined as a means to encapsulate and hide security details from software engineers. The basic exception approach (BEA) in Section 4.1 encapsulated URBS enforcement code but provided only limited software reuse, since the security code was pervasive throughout each class. The generic exception approach (GEA) in Section 4.2 solved these problems through the introduction of a generic security class that maintained encapsulation of URBS enforcement code while simultaneously facilitating software reuse. Advanced exception handling techniques will allow a wide-degree of versatility in providing corrective and notification code when security violations are detected, as presented in Section 4.3.

Two future areas of work that we have previously reported on are relevant for this effort [5]. First, the inability to utilize generics as desired in Section 3.2 raises again the need to open a dialog between researchers and practitioners in security, programming language, and compiler fields. Improvements and enchancements

to the runtime environments of languages like C++ are needed to successfully support URBS. Specifically, to eliminate code dependencies on condition and switch statements, the runtime environment of the C++ compiler must be changed to take into account the role of who's executing the code when determining which method to call, which is similar to dispatching in current compilers. URBS must be promoted as a first-class citizen during specification, design, and development, and its inclusion into languages and compilers would facilitate this consideration.

Second, URBS for object-oriented systems must be expanded to consider other languages like Ontos C++, Ada95, Eiffel, and JAVA. Expansion of URBS to Ontos C++ will transfer URBS concepts to a C++ compatible database platform, allowing the integration of programming and database concepts to be explored. The similarity of programming language and abstraction features in object-oriented languages means that a solution to URBS in one language will be transferable to other languages. For example, templates in C++, generic packages in Ada95, and generic classes in Eiffel, are similar concepts. As security on the WWW is of increasing concern, URBS solutions for JAVA applications will also need to be explored.

References

[1] J. Barkley, "Implementing Role-Based Access Control Using Object Technology", *Proc. of First ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, November 1995.

[2] S. Demurjian, M.-Y. Hu, T.C. Ting, and D. Kleinman, "Towards an Authorization Mechanism for User-Role Based Security in an Object-Oriented Design Model", *Proc. of 1993 Phoenix Conf. on Computers and Communications*, Scottsdale, AZ, March 1993.

[3] S. Demurjian and T.C. Ting, "The Factors that Influence Apropos Security Approaches for the Object-Oriented Paradigm", *Workshops in Computing*, Springer-Verlag, 1994.

[4] S. Demurjian, T. Daggett, T.C. Ting, and M.-Y. Hu, "URBS Enforcement Mechanisms for Object-Oriented Systems and Applications", in *Database Security, IX: Status and Prospects*, D. Spooner, S. Demurjian, and J. Dobson (eds.), Chapman Hall, 1995.

[5] S. Demurjian, M.-Y. Hu, and T.C. Ting, "Role-Based Access Control for Object-Oriented/C++ Systems", *Proc. of First ACM Workshop on Role-Based Access Control*, Gaithersburg, MD, November 1995.

[6] K. El Guemhioui, S. Demurjian, T. Peters, and H. Ellis, "Profiling in an Object-Oriented Design Environment that Supports Ada 9X and Ada 83 Code Generation", *Proc. of 1994 TriAda Conf.*, Baltimore, MD, Nov. 1994.

[7] H. Ellis and S. Demurjian, "Object-Oriented Design and Analyses for Advanced Application Development - Progress Towards a New Frontier", *Proc. of the 21st Annual ACM Computer Science Conf.*, Feb. 1993.

[8] W. Harrison and H. Ossher, "Subject-Oriented Programming (A Critique of Pure Objects)", *Proc. of 1993 OOPSLA Conf.*, Oct. 1993.

[9] M.-Y. Hu, S. Demurjian, and T.C. Ting, "User-Role Based Security Profiles for an Object-Oriented Design Model", in *Database Security, VI: Status and Prospects*, C. Landwehr and B. Thuraisingham (eds.), North-Holland, 1993.

[10] M.-Y. Hu, "Definition, Analyses, and Enforcement of User-Role Based Security in an Object-Oriented Design Model", *Ph.D. Degree Dissertation*, The University of Connecticut, May 1993.

[11] M.-Y. Hu, S. Demurjian, and T.C. Ting, "Unifying Structural and Security Modeling and Analyses in the ADAM Object-Oriented Design Environment", in *Database Security, VIII: Status and Prospects*, J. Biskup, C. Landwehr, and M. Morgenstern (eds.), Elsevier Science, 1994.

[12] F. H. Lochovsky and C. C. Woo, "Role-Based Security in Data Base Management Systems", in *Database Security: Status and Prospects*, C. Landwehr (ed.), North-Holland, 1988.

[13] D. Needham, S. Demurjian, K. El Guemhioui, T. Peters, P. Zemani, M. McMahon, H. Ellis " ADAM: A Language-Independent, Object-Oriented, Design Environment for Modeling Inheritance and Relationship Variants in Ada 95, C++, and Eiffel", *Proc. of 1996 TriAda Conf.*, Philadelphia, PA, December 1996.

[14] H. Ossher, et al., "Subject-Oriented Composition Rules"", *Proc. of 1995 OOPSLA Conf.*, Oct. 1995.

[15] F. Rabitti, et al., "A Model of Authorization for Next Generation Database Systems", *ACM Trans. on Database Systems*, Vol. 16, No. 1, March 1991.

[16] J. Richardson and P. Schwarz, "Aspects: Extending Objects to Support Multiple, Independent Roles", *Proc. of 1991 ACM SIGMOD Conf.*, May 1991.

[17] R. Sandhu, et al., "Role-Based Access Control Models", *IEEE Computer*, Vol. 29, No. 2, Feb. 1996.

[18] D. Spooner, "The Impact of Inheritance on Security in Object-Oriented Database Systems", in *Database Security, II: Status and Prospects*, C. Landwehr (ed.), North-Holland, 1989.

[19] T.C. Ting, "A User-Role Based Data Security Approach", in *Database Security: Status and Prospects*, C. Landwehr (ed.), North-Holland, 1988.

[20] T.C. Ting, S. Demurjian, and M.-Y. Hu, "Requirements, Capabilities, and Functionalities of User-Role Based Security for an Object-Oriented Design Model", in *Database Security, V: Status and Prospects*, C. Landwehr and S. Jajodia (eds.), North-Holland, 1992.