## Information Engineering: Object-Oriented Design and Analyses*

### Chapter 6: Object-Oriented Design and User-Role Based Security

Steven A. Demurjian, Mei-Yu Hu, T.C. Ting, and Thomas A. Daggett
Computer Science and Engineering Department
The University of Connecticut, Storrs, Connecticut 06269-3155

---

**Contents**

1. **URBS: Concepts and Issues** ........................................... 3
2. **Object-Oriented Design Model Enhancements** ......................... 5
3. **URBS Example for SDEs** .................................................. 10
   3.1 The SDE Example: OTs and RTs .................................. 11
   3.2 The SDE Example: URDH ............................................. 11
   3.3 Notation: Extending PPI Concepts ................................. 13
   3.4 URDH: Assigned Methods ........................................ 13
   3.5 The Privilege Revision Process .................................. 14
   3.6 Last Thoughts on SDE Example ................................... 16
4. **Security Issues for Object-Oriented Paradigm** ....................... 16
   4.1 From ADTs to the Object-Oriented Paradigm ..................... 17
   4.2 What’s in an Object-Oriented Application? ....................... 17
   4.3 The Impact of Advanced Object-Oriented Features ............. 18
   4.4 Claims of the Object-Oriented Paradigm ......................... 19
   4.5 Last Thoughts on Security Issues ................................. 20
5. **Security Code Generation and ADAM** ................................... 20
   5.1 Goals for a URBS Enforcement Mechanism .......................... 21
   5.2 Approaches for a URBS Enforcement Mechanism .................. 23
      5.2.1 The Brute-Force Approach .................................. 24
      5.2.2 The User-Role-Subclassing Approach ....................... 25
      5.2.3 The URDH Class Library Approach .......................... 28
      5.2.4 Comparing/Contrasting the Three Approaches ............. 31
6. **Advanced Security Code Generation** .................................. 32
   6.1 The Generic User-Role Subclassing Approach ..................... 32
   6.2 A Basic Exception Approach ...................................... 35
   6.3 A Generic Exception Approach ..................................... 37
   6.4 Advanced Exception Handling ...................................... 40
7. **Looking Ahead and Further Readings** .................................. 41

---

*Copyright ©1995 by S. A. Demurjian, Storrs, CT.*
Security has been a part of computing since the earliest days when systems became commercially available, with early systems had locks/keys for turning them on and off. With the advent of batch and interactive time-sharing systems, and the need for system managers to control access to these prized (and expensive resources), security came to the forefront of computing arriving at its pervasive presence today. Security approaches embody a broad spectrum, including:

- **OS Level** - Passwords to control access to time-shared systems.
- **En(De)cryption** - Public/Private key approach that publishes public encryption keys to allow individuals to send encoded messages that can only be decoded by the correct recipient who holds the private decryption key.
- **Mandatory Access Control** - Method favored by government agencies that tags information with levels (e.g., top-secret, secret, classified, etc.) and allows users to access information at or below their assigned level.
- **Discretionary Access Control** - Method intended to facilitate access to information by providing the means to discretionarily establish privileges for users that are customized based on their overall needs and responsibilities.

In today’s information dominated society, it is critical that access be controlled. Confidentiality of employment data, medical records, credit histories, and so on, must be protected to insure that individual rights to privacy are attained. Correspondingly, information access must be facilitated, so that the allowed individuals can access protected data in a timely fashion, allowing doctors to save lives, home mortgages to be processed in days or even hours, and so on. This dichotomy of confidentiality versus availability is what makes the attainment of fully secure and fully accessible applications/systems so difficult.

The emphasis that will be examined in this chapter is on discretionary access control, DAC, with a user-role based security (URBS) approach establish privileges for accessing object-oriented applications. Our unit of security will focus on object types and classes. Recall that most object-oriented programming and database languages have a single public interface (a set of public methods) that is shared by all users of an OT/Class. Consequently, the public interface can be characterized as the union of all possible methods required by all likely users. Moreover, if a method is placed in the public interface of a OT/class for one user, it is immediately available to all users, regardless of the original intent. This opens the possibility for misuse and corruption of information. For example, in HCA, a method to Prescribe_Medication that has been placed in a public interface for use by a Physician is immediately available to all other users (or tools) that have access to the involved public interface, which is a situation that is not desirable! Discretionary access control for object-oriented systems and applications seeks to customize access to the methods of the public interface at a type level. This will allow different portions of the public interface to be available to particular users at different times depending on their overall needs and responsibilities, which has been referred to a their user roles. Thus, the public interface for an OT/class is promoted as a “potential” public interface, representing all of the methods that may become public, but in a controlled manner to all potential users.

In addition to the HCA example previously given, there are many situations when an OT library designer (IDE - information design engineer) could utilize more fine-grained control to the public interface. For example:
• In software development environments, the public interface for Modules has methods that read
(for IDevEs - information development engineer and IManE - information management engineer)
and modify instances (only for IDevEs).

• In HTSS, the public interface for Items has methods that read (for Scanner, I-Controller) and
modify instances (only for I-Controller).

• In HCA, different health care professionals (e.g., Nurses vs. Physicians vs. Administrators, etc.)
require select access to sensitive patient data.

There is also a tradeoff between developers and end-users. During design and development, IDEs have
different roles based on their responsibilities related to cooperative design on an application, while
IDEs should only see those portions of the application that they need to see or that they will be
responsible for implementing. When an application is released, the end-users must be limited in their
interactions and access depending on their roles.

The user-role based security (URBS) approach to be presented in this chapter involves concepts
related to security that have been presented in Chapters 3 and 4. In Chapter 3, the Security section
of a specification contained the user roles for an application, while the ERCU UT-UR Deck of Chapter
4 introduced the concepts of user types, roles, and privileges based on who can, can't, or may, utilize
the responsibilities of classes. The material in this chapter builds on the user types and user role
concepts, by presenting a user-role definition hierarchy as an abstraction mechanism to support the
design and implementation of security privileges. To establish privileges for user roles, methods will be
both assigned and prohibited. An assigned method is a positive privilege that signifies that a given user
role can invoke the method. A prohibited method is a negative privilege that indicates that a given
user role can't invoke a method. Once privileges are defined, the application and its resulting
software must enforce them to ensure that information is correctly accessed by the right individuals at
all times. The URBS approach that is presented in this chapter intentionally obscures information
and its access for two reasons. First, we are interested in maintaining consistency with object-oriented
principles on encapsulation and hiding. This forces IDEs to focus on abstract concepts rather than on
detailed data access when defining privileges. The result is the incorporation of URBS into the object-
oriented design model in a manner that is consistent with the principles, precepts, and philosophy of
the object-oriented paradigm.

The remainder of this chapter investigates the various issues related to user-role based security and
the object-oriented paradigm, both in general, and specific to the ADAM environment. In Section 1, the
various concepts related to user-role based security are presented. Section 2 details the enhancements
that are needed to the object-oriented design model, presented in Chapter 5, to support user-role based
security, using HCA. To more completely examine the privileges definition and acquisition process
using URBS, Section 3 presents and analyzes an example from the domain of software-development
environments (SDEs). Given these general discussions as a context, Sections 4 and 5 consider the
realization of user-role based security within object-oriented systems, discussing both paradigm issues
(Section 4) and basic code generation mechanisms for ADAM (Section 5). Section 6 examines advanced
security code generation techniques that utilized generics and exception handling. Finally, Section 7
provides a summary and sets the context for later chapters.

1 URBS: Concepts and Issues

A user-role definition hierarchy (URDH), is used to establish and enumerate the possible user roles, as
the ISecE (information security engineer) attempts to characterize the different kinds of individuals who
will require access to a given application. We believe that regardless of the application, it is possible
to identify groups of individuals, that have related characteristics despite their different responsibilities
within the application. We also believe that individuals (or groups) will require different levels of access
to the application, ranging from the general to the specific. Hence, we employ a hierarchy to describe
responsibilities within an application and to represent associations between individuals (or groups) with
different, yet related, needs.
We have chosen to characterize the responsibilities of individuals within an application into three distinct levels of abstraction for the URDH, as discussed in Chapter 5: user roles, user types, and user classes. User roles are intended for the ISecE to characterize more specific responsibilities within an application. As an analogy, a user role would correspond to a specific job function of an employee such as grade-recording or transcript-issuing in the registrar's office of a university. The two roles are different; a grade-recording role enters changes and checks corrections, and can only inspect specific affected portions of the transcript; a transcript-issuing role will have different access.

When one or more user roles are related, we group them under a single user type, to represent similarities that exist among roles. In the example, the two user roles grade-recording and transcript-issuing can be grouped under the user type registrar-staff, since they all work in a common occupation. User types are a general level of abstraction, but as the example showed, still involves specific responsibilities that are common among the related user roles, e.g., all registrar-staff might share a common responsibility such as accessing names and addresses in student records. User types are also intended for the ISecE to assign particular privileges. In addition, since one or more user roles may be grouped under a user type, the privileges assigned to a user type are systematically passed to all appropriate user roles.

Finally, the third level of abstraction, user classes, is more general, and is intended to support responsibilities from a more global perspective. In our approach, the different user types of an application can be grouped into one or more user classes. For a university, appropriate user classes might be non-academic-staff (with user types such as purchasing-staff, campus-police, maintenance-staff, etc.) and academic-staff (with user types such as dept-staff, registrar-staff, presidents-office, etc.). In this example, the responsibilities are broadly divided into two user classes, which indicates that the ISecE intends to supply appropriate privileges to each class which are then passed on to the user types and to the user roles. The grouping of user types into user classes is very application-dependent, and to be correctly specified, requires the ISecE to fully understand the intent of the application. A sample URDH for a university application is given in Figure 6.1.

There are four key ideas to take from this brief and motivational example. First, the privileges that have been assigned to URs, UTs, and UCs have a specific flow down the URDH, allowing "lower" nodes to acquire privileges from "higher" nodes. In particular, privileges assigned to a UR are for its use only; assigned to a UT are passed to its URs; and, assigned to a UC and passed to its UTs and their URs. This leads to the key second idea, the employment of object-oriented concepts for the security definition process. Inheritance is used to clearly indicate how privileges are passed down through the URDH. Specialization lets an ISecE refine his/her design from UCs to UTs and from UTs to URs, while generalization can be utilized in the other direction. The third key idea is that the same UR under different UTs may have different meaning, which simply enforces the concept that the context of a given UR fully defines its capabilities. A related aspect of this idea is that URs cannot be further specialized. Instead, the fourth key idea allows the granularity of the UR to be designer definable, able
to represent different degrees of abstraction based on an applications needs. In HCA, in addition to
URs for Staff, RN and Manager, URs for TakeVitals and SetIV may also be appropriate.

2 Object-Oriented Design Model Enhancements

This section contains the extensions to the object-oriented design model presented in Chapter 5 to
support user-role based security. Like the earlier chapters, the key concern is to make sure the
extensions are consistent with the design model, in particular, and the object-oriented paradigm, in
general. The main focus of the chapter will be to identify the relevant profiles that are needed to define
security privileges. As each profile is explained, the emphasis is on the definition of privileges and the
acquisition of privileges that automatically occurs. Conceptually, privileges that are defined for a given
user role, type, or class, have an impact on other roles, types, and classes in the URDH. Also, privileges
that are explicitly defined form a subset of the entire privileges for a user role, type, and class, based
on a number of factors which will be discussed. Privilege definition and acquisition lead to a discussion
on methodological issues for URBS. In addition to identifying relevant profiles for URBS, changes to
existing profiles defined in Chapter 5 will also be detailed. Finally, this section also briefly explores the
automatic analyses that are available to assist IDEs and ISecEs in the security definition process.

To support the discussions in this chapter, both a set of OTs and a URDH for HCA are needed. In
Figure 6.2, we include OT descriptions for Item, Visit, Prescription, Test, Record, and Medical_R. For each of these OTs, we have indicated both the private data and the PPI methods. Much of the
information in this figure has been motivated and developed from examples in [44]. We have included a
wide range of basic methods that create and retrieve the different information. For clarity, each of the
methods is numbered with M.X.Y. The X refers to an OT. The Y refers to a method within an OT. We
have omitted the method profile for each method. Given this OT sub-hierarchy, in Figure 6.3, a URDH
for the health care application has been provided. From a top-down perspective in Figure 6.3a, there
are eight different user types: Nurse, Physician, Pharmacist, Technician, Therapist, Support, Patient,
and Spouse. In this case, the ISeCE is assuming that each of these user types may have privileges
that would be common to all user roles under the type. Within each user type, one or more user roles
may be defined. For example, in Figure 6.3b, user roles for Nurse include StaffRN, Discharge_Pling
(planning), Education, and Manager. In this top-down examination, specialization identifies the various
users and their roles. Figure 6.3a can also be examined from a bottom-up perspective to determine the
common characteristics by the grouping of the user types into user classes Medical, Staff, Support, Staff,
and Other. This is one possible grouping of user types. The bottom-up perspective corresponds to
generalization for establishing the user classes, as shown by the dashed lines.

To more accurately characterize the capabilities of user roles in the URDH, with respect to the
privileges to be granted against the application, we propose the creation of a node profile. A node
profile is a specialized base profile (see Definition 6 again) that can be defined as follows:

Defn. 14: A URDH node profile (NP) is a specialized BP (see Definition 6 again) that contains:

1. a name (from BP) for the node (UR, UT, or UC)
2. a prose description (from BP) of its responsibility
3. a set of assigned methods (the positive privileges)
4. a set of prohibited methods (the negative privileges)
5. a set of consistency criteria for relating URDH nodes

Node profiles are in turn specialized to characterize user class, user type, and user role security capa-
bilities, in the following definitions:

Defn. 15 and 16: A user-class (type) profile (UCP/UTP) is a specialized NP.

Defn. 17: A user-role profile (URP) is a specialized NP that also contains a prose description of its
security requirements.
<table>
<thead>
<tr>
<th>Item</th>
<th>Physician_Name</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1.1 Get_Phill_Name()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1.2 Set_Phill_Name(Name)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1.3 Get_Date()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M1.4 Set_Date(Date)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Visit**
- Symptom
- Diagnosis
- Treatment

**Prescription**
- Prescription_No
- Pharmacist_Name
- Medication

**Test**
- Test_Code
- Specimen_No
- Status
- Technician

| M2.1 Get_Symptom() | M3.1 Get_Prescrip_No() |
| M2.2 Set_Symptom(Sym) | M3.2 Set_Prescrip_No(No) |
| M2.3 Get_Diagnosis() | M3.3 Get_Pharmacist_Name() |
| M2.4 Set_Diagnosis(Diag) | M3.4 Set_Pharmacist_Name(Name) |
| M2.5 Get_Treatment() | M3.5 Get_Medication() |
| M2.6 Set_Treatment(Treat) | M3.6 Set_Medication(Med) |
| M4.1 Get_Test_Code() |                              |
| M4.2 Set_Test_Code(Code) |                              |
| M4.3 Get_Spec_No() |                              |
| M4.4 Set_Spec_No(No) |                              |
| M4.5 Get_Status() |                              |
| M4.6 Set_Status(St) |                              |
| M4.7 Get_Technician() |                              |
| M4.8 Set_Technician(Name) |                              |

**Record**

<table>
<thead>
<tr>
<th>Record_No</th>
</tr>
</thead>
<tbody>
<tr>
<td>M5.1 Get_Record_No()</td>
</tr>
<tr>
<td>M5.2 Set_Record_No(No)</td>
</tr>
<tr>
<td>M5.3 Get_Patient_Name()</td>
</tr>
<tr>
<td>M5.4 Set_Patient_Name(Name)</td>
</tr>
<tr>
<td>M5.5 Get_First_Date()</td>
</tr>
<tr>
<td>M5.6 Get_Last_Date()</td>
</tr>
</tbody>
</table>

**Medical_R**
- Medical_History
- Physical_Exam

| M6.1 Read_Med_Record() | M6.7 Get_Med_Hist() |
| M6.2 Insert_Visit(Visit) | M6.8 Insert_Phy_Exam(Exam) |
| M6.3 Get_Visit(Date) | M6.9 Get_Phy_Exam() |
| M6.4 Insert_Lab_Test(Test) | M6.10 Get_All_Visit() |
| M6.5 Get_Test(Date) | M6.11 Get_All_Medicine() |
| M6.6 Insert_Med_Hist(Hist) | M6.12 Get_All_Physician() |

Figure 6.2 Selected Object Types and their Private Data/PPI Methods.
Given these definitions, each of the different aspects of node profiles can be examined in turn.

Node descriptions are utilized to specify the responsibilities of a URDH node via a concise prose statement. Sample node descriptions for the HCA URDH given in Figure 6.3 are shown in Figure 6.4. Each of these descriptions can be supplied as the ISecE is creating the URDH and its roles, types, and classes, and may be refined/modified as needed.

Once the URDH has been specified, the ISecE assigns methods to hierarchy nodes, to characterize the privileges based on his/her understanding of the security requirements. We have focused on the methods that are assigned from the application (see Figure 6.2), with the data and its access intentionally obscured [42]. In [29, 30], implication rules compute implicit authorization which is similar to our process of method assignment, but differs since they assign objects and authorization types to roles. Our approach also contrasts to [26], where the access rights/ permitted roles are assigned based on data levels. Finally, our approach and [35] both have the goal of providing different interfaces to different users, but differ since they assign views based on data.

To support the method assignment process, the information required by each user and the associated privileges (e.g., read, write, or both) must be understood. This corresponds to the application’s intended security requirements, and is developed by the ISecE as part of the requirements definition for the application under development. We use the term access to mean read only. Explicit write needs are noted separately. Also, the term clinical information is broad, and coincides to Medical Records, Visits,
Prescriptions, and Tests (all information on a patient). Figure 6.5 contains a list of the role security requirements for HCA. Using this information, the ISecE can establish privileges for URDH nodes.

To illustrate the method assignment process, we highlight some possible assignments for the URDH in Figure 6.3 against the health care database of Figure 6.2. First, consider the user roles of Nurse: Staff.RN, Discharge.Ping, Education, and Manager. Individually, they all need read/write access to clinical information on patients. Staff.RN would likely have access to most Get methods from Figure 6.2 (14 methods), including, for example, Get.Symptom of Visit, Get.Medication of Prescription, Get.Patient.Name of Record, and Get.Test of Medical.R, but may be unable to access the Get.All methods. Staff.RN would also access the two read methods and a selected subset of the Set/Insert methods, e.g., Set.Symptom (record symptoms on patients), Set.Test.Code (record test to be conducted), Set.Patient.Name, Insert.Visit, Insert_Med_History (taken by nurses), etc. Discharge_Ping and Education nurses would have similar Get access, but more restricted write access, say Insert.Visit or Insert_Med_History, since the information accessing requirements state that they only write notes on patient progress. Managers would have all of the access of Staff.RN, but may also have additional access to fulfill their role (such as the Get.All_Medicine and Get.All_Protocols methods). Other assignments can involve the entire PPI of an OT. For example, the user type Nurse could be assigned all methods of Record, since all of its roles can access all of these methods. The role Physician/Private could be assigned all methods of Medical.R, indicating that the doctors are able to invoke all of the methods which are defined (M6.1 to M6.12). A better assignment for this role would be the methods from both Record and Medical.R, which doctors should also be able to utilize, e.g., doctors need access to the patient's name and visit history that is maintained in Record.

From the above discussion, a methodology for method assignment begins to appear. All privileges are assigned at the bottom-most level of the URDH, corresponding to the user roles. Given the privileges assigned to roles, the ISecE can examine the roles under a single user type (in this case, Nurse), and seek to identify commonalities between the specific assignments. Commonalities with respect to shared privileges are pushed up the URDH from the user roles to their shared user type. In this case, all of the Get methods and any common Insert methods can be moved up to the Nurse type. When multiple user types have the same user class (e.g., Nurse and Physician are under Medical.Staff in Figure 6.2), common methods may also be moved up to the user class. However, the methods must be common to all user types of a specific user class, or a type/role may acquire methods on which privileges were not intended. For example, all user types of Medical.Staff would have access to the Get.Patient.Name method of Record. A method that is shared by all user types of a URDH can be moved up to Users. Get.Patient.Name may be a method for this application. Thus, the specification methodology indicates that method assignments at the user roles flow up the URDH to the user types, user classes, and User. Flow of common information up the tree effectively forces differences in method assignments to be pushed down the tree.

The prohibited methods on a URDH node represent those methods which cannot be accessed by the URDH node. Thus, the ISecE augments the positive actions of the node (i.e., the assigned methods) with the non-allowed actions (i.e., the prohibited methods), as reflected in the role security requirements of the user roles (see Figure 6.5 again). Our concept of prohibited methods is similar to the concept of denied roles in [26] and permission tags in [4], but differs since both of their efforts emphasize data; we focus on types/methods.

Prohibited methods are very important in the overall specification of privileges. Recall from Definition 8 in Chapter 5, that a method profile contains, the methods which a method calls. Thus, there is the potential to call a great number of different methods which are defined on many different object types and that involve numerous private data items, when a single method is assigned to a URDH node. That is, when a user role is explicitly assigned a set of methods, the actual set that is available to the user role is the union of all assigned methods (from the user role, its type, and its class) and the methods that the assigned methods call, and so on. Since this is the case, the prohibited methods can be utilized to explicitly identify which methods cannot be accessed by a URDH node. That is, when an ISecE is doing design, s/he might assign a method to a user role that calls a method down-the-line that was supposed to be prohibited. Without the explicit support of prohibited methods, it would be impractical to assume that all methods not assigned are prohibited. With both the assigned and
prohibited method information, the analyses can automatically inform the ISecE when a prohibited method conflicts with a assigned method (or a method called by a assigned method, and so on) on a URDH node.

In our example, the ISecE can explicitly list the methods that the different user roles cannot access. For different roles of Nurse: Staff.RN cannot access Set.Treatment of Visit, Set.Medication of Prescription, Get.All methods from Medical.R, and so on; Discharge.PCng and Education would have a larger exclusionary list; and Manager would be similar to Staff.RN, but may be able to utilize some of the Get.All methods that Staff.RN cannot. A similar specification methodology to the steps described for assigned methods is also employed for prohibited methods. Common prohibited methods will pass up the URDH via the paths from user roles to a user type, and from user types to a user class. However, in this case, the prohibited methods are not explicitly repositioned in the URDH. Rather, the semantics of prohibited methods imply that a method prohibited from a user role (type), is also prohibited from its associated user type (class).

Consistency criteria in a node profile relate any two user roles, types, or classes with respect to their capabilities. Equivalence criteria allow the ISecE to identify which user roles (types/classes) must have the same capabilities, as reflected in the methods, OTs, private data that are assigned/prohibited. Equivalence criteria are very important for defining URBS, since whenever a change is made to the URDH (assigned/prohibited method is added/removed), the ISecE can be alerted that the privileges are no longer equivalent and particular nodes must be also modified. For example, from the role security requirements in Figure 6.5, the roles Physical, Respiratory, and Occupational of the Therapist user type, would all be equivalent, and a change to one role, would require a corresponding change to the other two.

Subsumption criteria allow the ISecE to establish an ordering among URDH nodes, indicating that the capabilities of one node cannot exceed the capabilities of another node. In our example, there are many subsumptions that should be specified by the ISecE. Both Education and Discharge.PCng are subsumed by Staff.RN (since the latter writes more portions of the database) which is subsumed by Manager. Education is also subsumed by Discharge.PCng, since the latter requires access to financial information. Physical, Respiratory, and Occupational roles are subsumed by Education, since the former three have limited database access. The three types of Therapist are also subsumed by Discharge.PCng, Staff.RN, and Manager, via transitive closure. Subsumptions at the user type level are also possible, e.g., Technician subsumed by Therapist. The different criteria are checked by comparing methods, OTs, or private data.

To assist the ISecE in the definition of privileges on the URDH via the node profiles, a set of new analysis techniques are provided. Some of these techniques are designer initiated; others automatically alert the designer to possible conflicts and inconsistencies. Designer-initiated analyses on a chosen URDH node are: a summary of the node descriptions including ancestors; a summary of the methods which have been assigned/prohibited (including ones acquired from ancestors); and, a summary of the method descriptions for assigned methods. Automatic analysis techniques are: the identification of conflicts between assigned/prohibited methods as reflected in the methods, OTs, or private data items of the application; checking the consistency of the different equivalences and subsumptions; and, alerting the designer when privileges given/removed to a chosen URDH node must also be made to other nodes to maintain existing equivalences or subsumptions. Also, since method calls can be nested, all of the analyses can be performed recursively to any desired level of depth.

By combining the different components of the URBS privilege definition process, a complete node profile for a user role, type, or class can be described, as shown in Figure 6.6. In this sample, the UR, the UR profile aggregates all of the components, with the key security privileges given in the assigned and prohibited methods. The assigned and prohibited methods that are shown represent the ones that have been explicitly assigned/prohibited to the user role, i.e., exclusive of the ones inherited from UT/UC.

URBS is supported within ADAM using one design phase and two semantic perspectives. In the single design phase, the URs, UTs, and UCs discussed in earlier chapters can be defined and organized into a hierarchy. In addition, privileges can be establish for each UR, UT, and UC. The two semantic perspectives provide complementary security analyses for the ISecE. One perspective allows the ISecE
to chose a node (UC, UT, or UR) and determine which privileges have been granted. A second perspective can be utilized by the ISecE to select a portion of an application (OT, method, private data/attribute) and determine which URs have been given access. Both perspectives are critical for insuring that security privileges meet the requirements as defined in the specification. Analyses will be more fully discussed in Chapter 7.

The actual mechanism for supporting the definition of URDH nodes in ADAM is accomplished via various dialog and display boxes. Initially, the ISecE must select the node type (user role, user type, user class) to define a new URDH node. After selecting the type of a node, the ISecE must supply the node name and node description for the created node. The initial information for the node profile of the user role Staff.RN is shown in Figure 6.7. After a node is created, the ISecE can select the menu option AddProfile to supply other information on the security privileges for a node, i.e., assigned and prohibited methods, and consistency criteria. The ISecE utilizes the mouse to select the assigned/prohibited methods from a list of previously defined methods. To specify the equivalence/subsumption criteria, the ISecE utilizes the mouse to select nodes from the list of defined URDH nodes. A selection of assigned methods for user role Staff.RN is shown in Figure 6.8.

In the URDH-specification phase of ADAM, the checking on assigned/prohibited methods is performed automatically to insure that there are no conflicts, e.g., an assigned method conflicts with an earlier prohibited method. If a problem is identified by the analyses, the system will not accept the specification and will require a correction by the designer. This is shown in Figure 6.9. Note that the conflict may be more subtle, and arise due to nested method calls, e.g., one assigned method calls a method that calls a method that is prohibited. The checking on consistency criteria is also performed automatically to insure that there are no conflicts when designers specify the assigned/prohibited methods and/or the consistency criteria. This is shown in Figure 6.10. The complete node profiles for the user roles Staff.RN and Manager, the user type Nurse, and user class Medical Staff, are given in Figure 6.11.

Finally, to close off this section, an updated definition of the application profile is given. (see Definition 11 in Chapter 5 again). This second version takes into account the need for the APP to contain URDHs:

**Defn. 11.v2:** An application profile (APP) is a specialized BP contains:

1. a prose description of the general purpose of the application (from BP)
2. a name for the application (from BP)
3. the OTs/RTs that comprise the application
4. the number of OTs/RTs in the application
5. a forest of URDHs

Remember, in practice, there are typically, at least two URDHs for an application. The first URDH, similar in concept to the ones given so far, represents the privileges for the different end-users that utilize the tools and capabilities of the application. A second URDH may also be required for the IEs that are involved in the design and implementation of an application. This second URDH can be utilized to enforce security privileges for IEs to insure that each IE is only allowed access to that portion of the application needed for his/her task, thereby control the information consistency of the application as it is designed and developed.

### 3 URBS Example for SDEs

To clarify the concepts related to the definition of the URDH presented in Section 2, a second example using software development environments (SDEs) is provided. This example includes the definition of OTs and RTs, as well as a URDH for different user roles for software design and development. To illustrate the URDH development process, the SDE URDH has been assigned a set of methods that have some potential conflicts related to the privileges that have been granted and/or omitted. Given this initial URDH, the analysis for security privileges is motivated by examining problems and postulating corrections.
3.1 The SDE Example: OTs and RTs

Figure 6.12 contains a possible conceptualization of software data is given, based on our work on SDEs [27] and similar to other approaches [6, 23, 33, 39, 46]. In this figure, from top-to-bottom, a Project contains one or more Systems, where each System contains one or more Modules. From bottom-to-top, the inverse relationships are maintained, i.e., a Module may be used by many Systems and a System may be used by many Projects. Collections of Projects, Systems, and Modules are also maintained (as indicated by the asterisks) to allow the searching of the instances of these three OTs independent of their associations. The SDECommon OT contains the shared features of the Project, System, and Module OTs, utilizing inheritance, and represented by the unlabeled arrows.

The remainder of the figure illustrates portions of the private data and the PPI for each OT, as separated by the horizontal line. Note that we have omitted many details such as method parameters, return types, methods in the API, etc. SDECommon contains the common features of a name, a set of users, a profile, a document, and methods for retrieving a profile or document, adding, deleting, and retrieving users, and storing a document. A profile is used to maintain historical and statistical information on the various aspects of a Project, System, and Module. Each OT has a specialized version of the Retr.Profile method, that will return different information depending on the type of the instance, i.e., an example of simple dispatching [48]. A document contains documentation on a Project, System, or Module.

The Project OT contains data for the name of the manager and methods for adding, deleting, and retrieving Systems, as well as methods for retrieving the profile and the lines-of-code. The System OT contains no private data and only has methods for adding, deleting, and retrieving modules, and for retrieving the profiles and lines-of-code. Module is where the detailed software data resides, namely, the source code, the object code, the parse tree, the symbol table, and a lines-of-code placeholder. Methods are included for adding, deleting, retrieving, and storing the different module representations and for retrieving the profile and lines-of-code.

The retrieve method for the lines-of-code is different for each OT: in a Module, it simply accesses the LOC data; in a System, it calls the Retr_Mod_LOC method for each Module of the System, and sums the intermediate results; in a Project, it calls the Retr_Sys_LOC method for each System and sums the intermediate results. The previous explanation appears to contradict Definition 1, which only permits the identification of intra-type methods. However, even though the method Retr_Sys_LOC calls Retr_Mod_LOC repeatedly, it does so to return information about the System OT, e.g., the lines-of-code for a System instance is calculated as the sum of the lines-of-code of all Modules that comprise the System. If the Retr_Sys_LOC method returned the lines-of-code for a module, then we would say that the method was inter-type, and not permitted under the restrictions of our design model, as given in Section 2.

For clarity, in Figure 6.12, we have numbered each of the methods with MX.Y. The X refers to an OT with 1 for SDECommon, 2 for Project, 3 for System, and 4 for Module. The Y refers to a method within an OT. M4.* refers to add (M4.3 to M4.6), delete (M4.7 to M4.10), retrieve (M4.11 to M4.14), and store (M4.15 to M4.18) methods for the source, object, symbol table, and parse-tree representations, respectively.

3.2 The SDE Example: URDH

To illustrate the aforementioned concepts more completely, in Figure 6, a number of user roles for a SDE are given. From a top-down perspective, there are five different user types, who are either software engineers (SE) or managers (M): a SE-Regular for implementation; a SE-Maintain for handling maintenance and software evolution; a SE-New that has just started in a company or on a project; a manager for a project, M-Project; and a manager for a division, M-Division. In this case, the ISecE is allowing for the possibility that each of these user types may have privileges that would be common to all user roles under the user type.

Within each user type, one or more user roles may be defined. SE-Regular may have user roles to Develop, Debug, Document, and Test software. SE-Maintain may have user roles for Document, Report_Bugs, and Fix_Bugs. SE-New may simply be allowed the user role of Learn, in order to be-
come familiar with the software of a project. MANAGERs, like SEs, also have different job types that depend on their ranking within the company. Roles for managers are also definable, such as Status (to get/set/modify the status for a Project or to get the status for a Division) and Report (the ability to generate reports on Projects/Divisions). All of these different user roles are intended to have particular privileges that reflect the responsibilities of the role within the application. In this top-down examination, we have utilized specialization for identifying the various users and their roles.

Figure 6.13 can also be examined from a bottom-up perspective to determine the common characteristics for the different user types. In this example, one possible grouping of the five user types into the user classes SE and MANAGER, is shown. Other groupings are possible, at the discretion of the ISeE. The bottom-up perspective corresponds to the process of generalization for establishing the user classes, as represented by the dashed lines. The user classes are defined by the ISeE to indicate that software engineers and managers may each share common privileges. This approach demonstrates a realistic use of multiple inheritance, since SE-Regular now inherits from two nodes, Users and SE, to fully define its characteristics.

Some other comments related to the URDH and this example must also be made. First, we must clarify that the same user role name under different user types has different interpretations, and therefore may have different privileges. Status has different interpretations under M-Project and M-Division, as noted above, which might require different privileges. Second, we have purposely not provided nodes in the hierarchy that would collect all of the SE user types under a single SE node and all of the MANAGER user types under a single MANAGER node, which in turn would be under the Users node and would add an extra level to the hierarchy. We strongly believe that the process of establishing user classes is one of generalization and not specialization. Since user classes depend on all possible user types, it would require that the application designer be able to predict all potential users in order to group them into user classes. In fact, the establishment of user classes may not occur when user types and user roles are being identified, but may be deferred until the latter stages of the design process.

Third, an argument could also be made to allow the user roles to be further specialized by supporting subroles, e.g., Test could be specialized to be Test_Module and Test_System. Nothing in the concepts we have presented precludes the identification of more fine-grained user roles, i.e., Test_Module and Test_System could have been user roles instead of Test. For now, we limit the depth of the URDH, thereby limiting the number of levels of abstraction, since we believe that added depth tends to significantly complicate the underlying concepts. Fourth, it should be clear to the reader that it is not possible to determine all of the needed user roles a priori. We expect that a ISeE would utilize a tool that allows them to create/modify URDH, as the needs of the application become clear and change over time. Finally, this example is not intended to be a complete characterization of all possible user types and user roles in a SDE. Rather, we simply illustrate how such a URDH may be defined.

Throughout the discussion, inheritance (via specialization and generalization) has played a major part in defining the roles, types, and classes, aligning with our goal of exploiting object-oriented concepts. For now, we are making the strong requirement that user classes cannot share any common user types and that user types cannot share any common user roles. Our intent is to limit the choices given to the designer, and to minimize the different types of abstraction which are supported. By not permitting an overlap of user classes or types, the designer must carefully establish the user classes and types for the application. We believe that the overlap of user classes and types has the potential to introduce both ambiguities and conflicts into the URDH. Further, since actual individuals will likely have multiple user roles in an application, overlapping can be supported by authorization, a topic which will be further discussed in Chapter 10.

3.3 Notation: Extending PPI Concepts

One unresolved issue concerns the technique for making the PPI available within type hierarchies. Once again, there is a tradeoff between database and programming language views of inheritance. One approach in databases mandates that only the OTs in the terminal leaves of an inheritance hierarchy are available for usage. This is true in the functional data model and Daplex [36]. If this approach is utilized, then only the PPI of the leaf OTs are relevant, and this interface will contain the aggregation of all public interfaces from the specific leaf to the root. Programming languages take a more general
approach, and allow usage to occur from any node in the inheritance hierarchy. To reconcile this conflict, the following definitions are supplied:

**LPI:** The *local public interface*, or LPI of the OT is the PPI at individual nodes in the inheritance hierarchy of an application, without considering inheritance.

**GPI:** The *global public interface*, or GPI, of an OT is the set of all LPIs including the current node through its ancestors from inheritance.

**API:** The *aggregate public interface*, API, of an application is the set of all possible potential public methods, namely, the union of all LPIs (i.e., all methods, regardless of type boundaries).

**UPI:** The *unified public interface*, UPI, of an application is \{LPL, GPI, API\}, the set of all possible public interfaces.

In the example of Figure 6.12, LPI for SDECommon has the methods Retr_Profile, Add_Users, Del_Users, Retr_User, Retr_Doc, and Str_Doc. Referring to the example again, the GPI for Project has the LPIs from both Project and SDECommon. All of these different public interfaces are available to the designer when the assignment of methods to user roles is occurring. The different interfaces are easily and automatically constructed and maintained while an application is being designed.

### 3.4 URDH: Assigned Methods

Once the URDH has been specified, the designer can start the process that assigns methods to specific nodes of the hierarchy. The designer is free to utilize the various public interfaces (i.e., LPI, GPI, API, or UPI, see Section 3.3) to make these assignments. Essentially, the actions taken by the designer are authorizing different user roles with permission to utilize certain public methods. *Utilizing* certain public methods means that an individual with a specific user role has permission to actually invoke the assigned methods.

To illustrate the concepts, Figure 6.13 is extended to include method assignments to URDH nodes, as shown in Figure 6.14. We utilize the method numbers presented in Section 3.3 to differentiate between the methods of the different OTs.\(^1\) The position of the method in the hierarchy indicates which other nodes acquire (and possibly pass on) the permission to utilize the method. Starting at the top, any method assigned to Users, is acquired by all of the user types which then passes on the method to all of the defined user roles in the hierarchy. In the method assignments of Figure 6.14, we have not assigned any methods to Users. Recall from Figure 6.12 that Project, System, and Module were all collections, and as such, also have a set of methods to manipulate elements of the collection (which were not provided in order to simplify the example). If collection methods did exist, the method that locates a project by name would be ideally assigned to Users, allowing all user types (and their user roles) to also be assigned that method. In this case, the ISecE is stipulating that all individuals, regardless of their role, need access to a locate project method.

In examining Figure 6.14, we observe that the method set \{LPI1, M2.3, M3.3\} is assigned to the user class SE. This method set contains all methods from SDECommon, Retr_System (M2.3) from Project, and Retr_Module (M3.3) from System. In this case, the ISecE is assuming that all user types of SE (i.e., SE-Regular, SE-Maintain, and SE-New) and their respective user roles will need access to this set. Also in the figure, SE-Regular has been assigned LPI4, since the ISecE believes that all roles under SE-Regual will require access to all Module methods. Finally, when methods are assigned to leaf nodes, then these methods are restricted to the given user role. The scope and rationale for each assignment is given below:

- Develop of SE-Regular has been given access to all of the System methods (LPI3) plus the Add_System (M2.1) and Del_System (M2.2) of Project. In this case, the ISecE has determined that in addition to LPI4 (acquired from SE-Regular) and methods acquired from the user class.

\(^1\)The subscripts 1, 2, 3, and 4, map to the OTs SDECommon, Project, System, and Module, respectively, for the LPIs and GPIs.
SE, a Develop role might also need to change the Modules in a System (via LPI3) or the Systems in a Project (via M2.1 and M2.2).

- Both Document roles have been assigned LPI1 (from SE) mainly to utilize Retr_User (M1.4), Retr_Doc (M1.5), and Str_Doc (M1.6) of SDECommon, and methods to retrieve the profile (M2.4 and M3.4) and lines of code (M2.5 and M3.5) for Project and System, respectively. In addition, Document under SE-Regular has been given permission to access all methods from LPI4 (via SE-Regular), to be able to fulfill its responsibilities.

- Fix_Bugs has been assigned all Module methods via LPI4 in order to identify and correct errors within a project. This role also acquires methods from the user class SE.

- Report_Bugs has been assigned limited access of Module via Retr_Profile (M4.1). In this case, the ISecE is stipulating that this user role needs minimal access beyond the methods acquired from the user class SE to satisfy its required responsibilities.

- Learn has been assigned Retr_Profile of Project (M2.4) and Module (M4.1). This is reasonable, since a new person would need limited access to project-related data in addition to the access provided by the user class SE.

In the remainder of the hierarchy, the MANAGER user class has been assigned all methods of SDECommon (LPI1). In this case, the ISecE has decided that all types and roles under MANAGER will require access to information in SDECommon via its methods. Finally, the M-Project user type has been given all methods of Project (LPI2). The ISecE has given broader power to the M-Project user type (and its user roles), since it is likely that project managers are more involved in day-to-day operations and therefore require additional access capabilities. In Section 3.5, we will see that these method assignments are flawed, and through analyses can be revised to more precisely satisfy the security needs of the application.

An important aspect of the above discussion is our continued exploitation of object-oriented principles in the definition of URBS. The assignment of methods to nodes coupled with the implied inheritance hierarchies, results in a clear and precise interpretation for the designer. A designer is assured that methods assigned to a given node will trickle down through the hierarchy and apply to all relevant node. If a method is only shared by a subset of the roles under a type, the designer simply assigns the method to each of the desired user roles.

3.5 The Privilege Revision Process

Once a baseline URDH has been established, the analyses that an ISecE can perform is utilized to refine and redefine security privileges. The goal is for the ISecE to identify and correct flaws in the security definition of the URs as realized in the URDH. To assist in the privilege revision process, the ISecE can initiate analyses from two complementary perspectives. First, by choosing a UR, UT, or UC in the URDH, the ISecE can determine the breadth of access given to a URDH node. Specifically, the ISecE can determine the OTs, methods, and/or private data that have been granted to the URDH node based on the assigned methods. The result of the analyses in this case can be utilized to discern if a chosen URDH node has been given too many privileges or has been denied access to a needed portion of the application. Second, by selecting an OT, method, and/or private data item in the application, the ISecE can determine the different URs that have been given access. The result of these analyses can be utilized to understand whether too many URs have been given too much access, or UR(s) were inadvertently denied access.

From the perspective of the URDH, consider the method assignment given in Figure 6.14. The MANAGER/M-Division node in Figure 6.14 inherits the methods LPI1 from the MANAGER user class. This assignment allows division managers to access all of the methods in the SDECommon (see Figure 6.12), namely the Add_User and Del_User methods. One analysis would allow the ISecE to recognize that the division managers have been given too much access according to the security requirements. A more appropriate assignment would contain M1.1, M1.4, and M1.5 from SDECommon.
and perhaps M2.4 from Project, limiting the access of division managers to methods that only read data. The analysis also allows the ISecE to identify when methods have been omitted from assignment. For example, the role SE/SE-Maintain/Document was not assigned methods to access Module, which is required to successfully document corrections to bugs. In this case, the ISecE can assign a subset of LP14 to this role.

In another type of URDH analysis, the ISecE can gauge and understand the implications of the method assignment from the perspective of the involved OTs, to verify whether (he) intended to provide (deny) access to a specific OT by a chosen node. As an example of the first case, the role SE/SE-Regular/Document has access to Module via LP14. In this case, the ISecE did not intend to give access to the entire OT for documentation, since there is no need to modify Modules. Revisions can be made by moving LP14 down the hierarchy to specific user roles that require access to all Module methods (Develop and Debug) and assigning a subset of LP14 to Document. In the second case, the user role SE/SE-Maintain/Document only has access to SDECommon, Project, and System. Since the security requirements likely stipulated that Module access is also need for this role, a subset of LP14 can also be assigned.

A third type of URDH analysis alerts the ISecE to conflicts in private data access. A conflict occurs when two or more methods both write the same private data item(s) or when the chosen node has unauthorized access. For example, in Figure 6.14 SE/SE-Regular/Document has access to the method Add Users of SDECommon. This allows the Document role to modify the UserSet private data item, which likely contradicts with the intended security requirements and would necessitate a change that removes relevant assigned methods. In the other case, a chosen node was not given access. For example, the Report Bugs role should have been given access to Source of Module, to report on how corrections in the code were made. In this case, access to M4.1 and M4.11 (to retrieve Source) can be supplied.

From the perspective of the application, complementary analyses can be performed whose results are useful to the ISecE in revising the URDH. As an example, consider SDECommon as the selected OT. A number of roles access SDECommon, in particular, MANAGER/M-Division/Report. During this analysis, the ISecE realizes that division managers should only be allowed to obtain status information, and should not be allowed to do reports, since this responsibility was not included in the description for this role in the security requirements. If this is the case, the Report role can be deleted from the hierarchy. Conversely, when the Module OT is selected, the ISecE finds that SE-Maintain/Document does not have access, and can revise URDH as described earlier to correct this oversight.

In another type of application analyses, when an OT is selected, the URs that can access the OT's methods are returned. For example, this analysis returns the fact that all of the roles of SE-Regular can use all of SDECommon's methods, namely, the Add User and Del User methods. If this was not intended by the ISecE, then corrections can be made that move LP14 down to relevant user roles and add in specific privileges to other user roles. Conversely, when the role Document does not turn up on any of the method lists for the Module OT, the oversight can be corrected by adding Module methods to the user role. The method for this analysis technique is defined as follows:

In a third type of application analyses, the access to private data, either by a single user role or a number of user roles in conjunction, can be investigated by the ISecE. For example, since LP11 is assigned to MANAGER, all of the user roles of this user class (specifically, M-Division/Report and M-Division/Status) appear on the UserSet private data item, with read/write access. To rectify this problem, the method assignments can be changed by removing LP11 and adding only those specific methods which are required (M1.1, M1.4, M1.5, and M2.4). Alternatively, a role may be missing from the private data list of the selected OT. For example, if the Document role of SE-Maintain is not on the private data list of Source when the Module OT is selected, the the Module methods M4.1 and M4.2 can be assigned.

To complete our example, Figure 6.15 contains a revised version of the URDH of Figure 6.14, with more realistic method assignments. There have been significant changes from the original URDH. The MANAGER user class has been limited to methods from SDECommon and Project that just read information, and these methods are available to all user types and their roles. This eliminates roles of M-Division having too much access. The M-Project user type has also been revised, with permission moved down to its roles, allowing access of SDECommon and Project by the Status role (via GPI2),

15
and additional read access by the Report role via the method Retr.Proj.LOC. Changes to the SE
user class have assigned only read methods, that are then available to its user types and their roles.
The assignment of only read methods to user classes (and even user types) is reasonable. Since these
methods are passed onto user roles, any methods that update information would likely be received
by a role that shouldn’t be given such a privilege. Another major change has moved LPI4 down the
hierarchy from the SE-Regular user type to only the roles which require access to all operations of
Module. This is since its original placement allowed the Document and Test roles to update Module
instances, which was neither desirable nor intended. Other changes in this portion of the hierarchy
include methods to read Module information for both the Document and Learn roles. The Document
roles have also been given the ability to use the method Str.Doc(M1.6) for saving documentation, and
the Develop role has been more clearly defined in its access to methods in Project and System.

3.6 Last Thoughts on SDE Example

In retrospect, the SDE example has many lessons that can be learned regarding the security definition
process. When viewed from the perspective of the actions taken by the ISecE to refine and redefine
the URDH, it becomes immediately evident that differences in privileges trickle downward (from UC
to UT to UR) while commonalities move upward. This is an excellent illustration of the use of both
specialization and generalization in the URDH. The SDE example also indicates the ways that analysis
can be utilized by ISecEs to identify cases when too many (few) privileges have been given. This
is crucial to guide the ISecE during this process for arriving at more accurate and precise security
definitions.

The SDE example also illustrates the possible URDH changes that can occur as an ISecE attempts
to reconcile problems. The easiest change is to method assignments, but it must be noted that a change
of assignments can cause problems elsewhere, especially when the change occurs to a UT or UC node.
Adding and deleting roles are useful when the given URDH doesn’t represent the intended domain.
When too many privileges are given, it may also be necessary for an ISecE to split a role into two
or more roles. Changes to the application are also possible. The majority of application changes are
 traced to: the addition/deletion of OTs, methods, and private data items; a change in the inheritance
structure among OTs; or a modification to the signature of a method. In all cases, the changes must
be carefully evaluated against the URDH to ensure that the resulting security privileges are consistent.

4 Security Issues for Object-Oriented Paradigm

As a result of the increased interest and utilization of the object-oriented paradigm in many diverse
areas of computing research, design, and development, researchers have begun to examine security issues
and considerations. There have been efforts on investigating mandatory access control (MAC) using
the Bell and LaPadula security model [3] to protect access to information by classifying and tagging
data, for object-oriented systems [24, 40]. User-role based security (URBS) [26, 41] for supporting
discretionary access control (DAC) to offer freedom and versatility in the privilege-granting process,
has also been explored at an instance-based level [4, 26, 30], that complements the work presented in
this chapter. Often many of these proposed solutions do not consider the unique and special features
of the object-oriented paradigm.

However, in extending existing approaches to security for object-oriented systems have we placed
the proverbial cart before the horse? Are we mandating that security for object-oriented systems must
take an existing approach, when instead we should be asking if there is a better or new choice that
can be made? Moreover, are there unique characteristics and features of object-oriented systems and
applications that can and should impact our approach to its security? Hence, it is appropriate to
consider the not-so-rhetorical question: Shouldn’t the object-oriented paradigm influence and guide the
approach for security? The remainder of this section addresses this important question by providing a
foundation for thought provoking discussion on the features of the paradigm that should impact and
determine the shape and form of its security.
4.1 From ADTs to the Object-Oriented Paradigm

The object-oriented paradigm, while receiving recent attention, is not new. Its foundations can be traced to the emphasis in the early 1970s on modular program development via abstract data types (ADTs) [25, 34]. When using ADTs, the key is to achieve representation independence, thereby presenting a straightforward interface to the ADT while simultaneously hiding the implementation. This allows implementation changes to be made with no impact on the interface and its use. At almost the same time, similar abstraction methods were also underway in the database area, led by the classical work on aggregation and generalization by Smith and Smith [37]. The promotion of inheritance by their work and later extensions to Chen’s entity-relationship model [5] were both important to transition from ADTs to the object-oriented paradigm.

Like ADTs, the object-oriented paradigm promotes the development of object types (OTs) or classes, where both the data and methods (operations) are encapsulated. For representation independence, the implementation is hidden, thereby controlling access to a type’s data and methods. Like ADTs, an interface provides the visible access capabilities. Inheritance offers a controlled sharing of data and methods between related OTs in an application. Object-oriented concepts have been covered in great detail elsewhere [45, 47]. How are these concepts related to security? Encapsulation, the binding of information (data) and behavior (methods), sets strict requirements on an OT, reducing allowable actions; a definite benefit if security is to control access to the type, with hiding playing a major role. In addition, inheritance shares information, which implicitly indicates that sharing from a security perspective is occurring. The conclusion is that any definition of security for object-oriented systems must begin with encapsulation, hiding, and inheritance, the three cornerstones of the paradigm.

4.2 What’s in an Object-Oriented Application?

While object-oriented applications vary widely in their functionalities and capabilities, there is growing evidence that the greatest potential of its usage lies in advanced and complex systems that are large scale, heterogeneous, and distributed. Examples of such domains include health care, manufacturing, and software environments. All of these domains need to maintain high volumes of disparate data persistently, and also involve numerous individuals with varied responsibilities and needs, that must work together in a collaborative and cooperative fashion within each system. For example, in the health care domain, data varies from images (MRI, X-ray, etc.) to patient records, providers include hospitals, outpatient clinics, physician offices, etc., who all must interact with the insurance industry, and governmental and other agencies. For this domain, it has been argued [41] that URBS is the most appropriate means to support security requirements via the definition of roles and content and context based security constraints. The intent is that this domain (and others) requires a fine-grained level of control to define and maintain privileges. How can the object-oriented paradigm support such a level of control? In fact, a better question is: What feature or characteristic of the object-oriented paradigm must be explored to provide such a capability?

In the discussion of the previous section, it is apparent that the public interface, the set of all visible methods, their parameters, and their return types for each OT or class, may be the first place to look to answer the previous questions. The public interface, once defined, provides the access means to the OT and its instances, but is limiting from a security perspective since all users, regardless of their needs within the application, have full access to all methods in the public interface. That is, the public interface contains the union of all of the methods that need to be public for all possible users, which unfortunately, permits everyone to have access to some methods that were only intended for a select one or two users. For example, in an OT or class that maintains the medical records of patients at a hospital, physicians must be given access to methods that set the medication or treatment, requiring that these methods be part of the public interface. If that is the case, then nurses, pharmacists, technicians, and other hospital staff will also have access, since once a method is public, it is available to all. To selectively grant access to the public interface, techniques must be devised that allow different individuals to have particular access to specific subsets of the public interface at different times, based on their roles within the application. For the previous example, access by non-physicians to methods that set medication/treatment can now be prohibited, i.e., can be selectively given only to physicians.
Thus, in addition to encapsulation, hiding, and inheritance, the customization of access to the public interface must also be a guiding factor in security for object-oriented systems.

In fact, there is an argument to be made that the instance characteristics of object-oriented applications also play a significant role in guiding the appropriate security approach. Most, if not all, object-oriented applications are similar in that they depend on the development of extensive OT or class libraries. These libraries often contain multiple inheritance hierarchies for sharing code and promoting reuse. The number of OTs/classes in libraries can be quite small (in the tens), but typically contain hundreds and even thousands of different types. During runtime, the actual number of instances is also widely varying. It is not unusual to observe that the majority of classes have only a few instances (low hundreds), with a select group of classes dedicated to manage collections or sets of instances. In fact, many applications contain numerous classes with very few instances (tens or less). Therefore, in practice, there are likely a large number of types, possessing a wide range of instances (very few to thousands or more). Complicating this diverse behavior is the fact that it is not unusual for instances to change types due to inheritance, or for a type to go from having very few instances to very many instances (and back again) over time. The above discussion seems to preclude the usage of MAC for object-oriented systems, which has traditionally shown strong ties and applicability to relational and other structured data. This data has very regular characteristics; a few types (in the tens), with (tens-of-)thousands of instances per type. While the overhead to support MAC in such data is manageable, it quickly multiplies for the expansive and extensible features of object-oriented systems. Perhaps we must look for another approach to deal with these issues; one more suited to the described application needs and requirements. Clearly, the type and instance characteristics of an object-oriented application will play a pivotal role in defining and supporting security.

4.3 The Impact of Advanced Object-Oriented Features

Advanced object-oriented features, namely, polymorphism, dispatching, and overloading, strongly influence the design and engineering practice for object-oriented software. Polymorphism is important in the paradigm since it allows type-independent software to be developed. For example, instead of defining different stack classes for different data types (e.g., one stack for integer, one stack for reals, one stack for strings, etc.), a single stack can be defined that can then be utilized regardless of the data type of the stack elements. Polymorphism is supported in object-oriented languages via generics or parameterized types, and strongly promotes software reuse. Dispatching is the run-time or dynamic choice of the method to be called based on the type of the calling instance. For example, an inheritance hierarchy for graphical objects would contain classes for Circle, Rectangle, Triangle, etc., all children of a Shapes parent class. Each class would have its own method to graphically display the shape. For simplicity, all shapes (instances of Circle, Rectangle, Triangle, etc.) would be treated as being a collection of their common parent (Shape). Despite this commonality, when the display method is invoked on an instance, the underlying type will result in specific (and different) methods being called at runtime. Dispatching offers many different benefits to the software engineer, all related to extensibility and productivity, including: more versatility in the development of inheritance hierarchies and class libraries; the promotion of software reuse and evolution; and, the ability to more easily and effectively develop and debug generic code. Lastly, overloading, is the ability to define two or more methods with the same name but different signatures. In all programming languages, operations like +, -, etc., are overloaded, since they can be used for integers, reals, sets, and so on. Both object-oriented and other programming languages allow users to overload these and other operations in their user-defined data types.

How can and should these three advanced features support security? Polymorphism, through its type independence of code, might be the vehicle by which security code for object-oriented systems can be successfully implemented and reused. In a URBs solution to security, different roles must all undergo the same processes of granting privileges, authentication, and enforcement. When establishing a security policy for an application, polymorphism can be used to develop class libraries for supporting these processes, that are parameterized by type (in this case, user role!). Dispatching and overloading are strongly linked, and together allow an executing piece of object-oriented code to behave differently based on the type of the invoking instance. There is a strong parallel from a security perspective;
dispatching and overloading have strong ties to promoting and supporting the execution of security code via the runtime invocation of different methods based on the involved user role. In this case, the security policy and its associated code can be extended and modified as needed when user roles (or their capabilities) change over time. The common theme of all three advanced features is geared towards class libraries that, by their nature, support software reuse, extensibility, and evolution, all of which must take a significant part in defining security for object-oriented systems.

4.4 Claims of the Object-Oriented Paradigm

There have been many different claims on the benefits and advantages of the object-oriented paradigm, ranging from the obvious and easily proved (stresses modularity) to the generally accepted realities (promotes software reuse and facilitates software evolution) to the difficult to verify, but hopefully true (controls data consistency and increases productivity). The claims that have the most impact on security for object-oriented systems involve software reuse and evolution, as discussed in Section 4.3. If these claims are true (in practice, many researchers and developers have achieved success), then these two important aspects of the paradigm must play a leading role in security design and development. These two claims are tightly linked to the definition and maintenance of OT/class libraries for object-oriented applications.

There are many possible scenarios that can form the basis of a solution. Clearly, security in object-oriented systems can be implemented through the design and development of OT or class libraries. Such an approach would be useful, whether the security is DAC, URBS, MAC, etc. Once defined, these libraries can be reused as is, extended with new capabilities, or evolved to satisfy changing needs. For a given application (like health care), as new software and tools are developed with an object-oriented approach, apropo security libraries would be included, based on the functional and security requirements for the software/tool. These libraries would provide all aspects of the security, such as definition, authentication, and enforcement. For example, referring again to the health care application of Section 2, suppose that a software tool to monitor and establish treatment was to be developed for all professionals that administer care, e.g., nurses, physicians, technicians, etc. In a URBS approach, each of these professionals would have different user roles. The overall security policy for such an application would need to consider and distinguish the security requirements for each role. If such a policy for health care was implemented as a class library, then the user role for physician would be given more expansive access to the library (to allow doctors to set medication and treatment) than nurses. When such a policy is included in the software tool, the end result is that the tool behaves differently based on the user and his/her role (dispatching again).

To realize the aforementioned scenario of a class library for security, where the same tool would operate differently depending on the user role, there must be support at the implementation level in the definition of OTs/classes. One way to provide such support would be to enhance and expand the capabilities of the constructor. A constructor for an OT or class is utilized to create an instance. This could be extended to also include instances of the relevant security classes, to define the security policy for the class. In this way, we begin to bridge the gap to transition from type-level security to its instance-level realization. Another choice would be to add a security constructor to an OT/class, that would specifically and uniquely embody the security policy. Extensions to a class are consistent with other work that has added integrity constraints and triggers to classes [1]. Regardless of the final choice, the idea of a security class library, and its inclusion and reuse both within and across applications, can be strongly advocated as consistent and in sync with object-oriented precepts and principles.

4.5 Last Thoughts on Security Issues

The intent of this section has been to open and promote discussion on the considerations and concerns that must be the guiding factors in the design and development of apropo security approaches for object-oriented systems. Our opinion is that such design and development must have strong foundations in fundamental (encapsulation, hiding, and inheritance) and advanced (polymorphism, dispatching, and overloading) object-oriented concepts, geared towards software reuse, extensibility, and evolution. There must be a concerted effort for a security approach that embodies an OT/class library solution.
(as argued in Section 4.4), providing the means to define and enforce an application’s overall security policy. Moreover, the approach must be versatile enough to allow different security requirements to be realized on specific OTs/classes or on entire inheritance hierarchies (see Section 4.2 again), based on user roles. Such an approach will allow software and tools to be made available that appear, react, and behave in a customized fashion, based on the user role, thereby dynamically enforcing the security requirements. However, it is important to note that the thoughts in this section constitute only a starting point. There are many other important issues that must be considered, including support for concurrent engineering via cooperative and collaborative work [7]. In addition, temporal issues for object-oriented models must also be explored, to allow different user roles access to an application at different times.

5 Security Code Generation and ADAM

Even though ADAM can generate code from object-oriented designs, its security capabilities are limited to the definition and analyses of user roles, as has been described in earlier sections. Thus, while URBS definition is supported, the defined security has not been realized in the generated code at any level. The work presented in this section investigates this next step in the process of designing and developing object-oriented applications by proposing and evaluating three alternative enforcement mechanisms that can be automatically generated by ADAM. Before doing so, there are a number of critical issues must be considered to design and develop a URBS enforcement mechanism for the object-oriented paradigm.

First and foremost, as we have argued in the previous section, any approach for supporting security definition and enforcement must be consistent with object-oriented precepts and principles. This includes: public/private interfaces, encapsulation, information hiding, inheritance, polymorphism, dispatching, overloading, software reuse, software evolution, and extensibility. A second issue is that the user roles associated with object-oriented applications are often very dissimilar, since the application itself may be utilized by a wide variety of individuals, each performing very different tasks as characterized by their user roles. For example, the health care application (HCA) from our previous work [20, 21, 22] aptly illustrates this issue. Doctors that assign prescriptions, nurses that take vital signs, personnel who check-in and bill patients, all have different roles and associated responsibilities, thereby requiring particular access to patient data. While commonalities exist across roles (nurses, doctors, and check-in/billing personnel all read patients’ name, address, etc.), differences are also clearly evident in the previous example.

A third issue is that security itself is often temporally impacted as the application goes through different phases of utilization, and as the user roles and the roles associated with an individual change with time. Clearly, the security policy and the mechanism that enforces it must be conducive to supporting these dynamic requirements. Thus, the design and development of a security enforcement mechanism will also be guided by its need for extension and evolution to meet changing and new requirements over time. If we can automatically generate “code” for an enforcement mechanism, then reuse is important to reduce required effort.

Fourth, any definition of a security enforcement mechanism for object-oriented systems and applications, must encompass many different perspectives and interactions that span the design, development, implementation, and utilization processes. The enforcement mechanism must handle the security policy at a type level, an instance level, and at a user level. The enforcement mechanism must be consistent with the concept of inheritance, from both an object-type definition (inherting of attributes and methods from classes to subclasses) and utilization of inheritance for extensibility and evolution (extending and/or evolving the application).

Finally, one of the more difficult issues we have faced is attempting to target emerging systems that combine programming and database capabilities. Object-oriented database systems that are C++ based have been our focus, since they tend to support object-oriented programming and software engineering. Our efforts have been a combination of programming, software engineering, databases, and security as we seek to provide a means to support URBS within this joint framework. URBS in an object-oriented framework must facilitate the development of application code against the persistent
class library that is supported by an underlying object-oriented database system. When an application is compiled, its role-based enforcement capabilities are embedded into the resulting executable image. When an application is executing, the user role of the logged in individual is utilized to provide the means to enforce security. Such an approach provides a seamless transition from design to development to use. It is important to note that today's object-oriented languages like C++ do not yet support some of the critical capabilities that are required to have a truly dynamic, extensible, and customizable URBS enforcement mechanism.

There have been a number of related efforts that we have examined. We have been influenced by [38], since it clearly identifies the different issues related to inheritance and security, which are critical for an URBS enforcement mechanism for object-oriented systems. The approach in [30] is relevant since it compares to our overall URBS effort, but doesn't consider the generation of a security enforcement mechanism as a part of the application; in their effort, its part of the database system which can cause problems if one attempts to fully support features in an object-oriented programming language. There has been work on aspects as a mechanism for object-oriented data models to extend a given class with new capabilities, including, roles [31]. Similar in concept to our work, in their approach, 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. In a later effort [32], capabilities and access control lists are utilized to allow the owners of objects to control who can invoke methods on which objects. This effort differs from our approach, since it offers at an instance level of control. Also, the inclusion of users/user roles into their model is by no means obvious. Another effort allows different classes in an application to have different subjective views [19], 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++ [28]. Finally, an effort on role-based access control for object technology [2] 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. All efforts [2, 19, 28, 31, 32], like our own work, examine the realization of their security approaches via generated code.

The main purpose of this section is to investigate URBS enforcement mechanisms for object-oriented systems and applications, in general, and as realized in the ADAM environment, in particular. There are a number of goals that are instrumental for evaluating the utility and effectiveness of the different URBS enforcement mechanisms. These goals are strongly related to the object-oriented paradigm, and include, extensibility, flexibility, hiding and encapsulation, and reusability. There are also three alternative approaches for a URBS enforcement mechanism, using the ADAM environment as a vehicle to illustrate generated code for each alternative. This section also includes a summary that compares and contrasts the alternative approaches against the goals.

5.1 Goals for a URBS Enforcement Mechanism

The goals for a URBS enforcement mechanism for object-oriented systems and applications can be defined, from two perspectives. From the first perspective, the goals must be capable of evaluating whether the enforcement mechanism meets the needs and requirements of an object-oriented application. For example, since an object-oriented application is always changing (new OTs, new inheritance relationships, new URTs, etc.), the enforcement mechanisms must be able to handle these changes. From the second perspective, the goals must be able to evaluate the enforcement mechanism itself, allowing us to gauge whether a given alternative exhibits certain characteristics independent of the application and/or object-oriented paradigm. For example, if new capabilities or security policies are developed, can the enforcement mechanism easily adapt to consider these changes, without requiring changes to the application. The following examines the goals of extensibility, flexibility, hiding and encapsulation, and reusability, from the two perspectives:

- **Extensibility:** "Nothing is ever complete!"

  Application Perspective: Extensibility is often required in the life cycle of an application, since new classes can be added, new methods can extend classes, new user roles
can provide for new responsibilities, new individuals can be authorized, and so on. Therefore, as the various components and aspects of an application are extended with new capabilities, the enforcement mechanism must be able to readily support these new extensions with a minimal impact (hopefully, none) on the overall development process.

URBS Enforcement Mechanism Perspective: An extensible enforcement mechanism is required, since it is possible that as the application is evolved over time, new security policies and approaches may be defined. While it would be nice if such a dramatic extension were possible, it seems that, at least at these early stages, such a capability is unattainable, i.e., its akin to changing a security policy from MAC to DAC. In theory, any mechanism must be conducive to the rapid and unexpected addition of new restrictions. In practice, extensions related to changes to the application are the most critical to support.

*Flexibility: “Change is inevitable.”*

Application Perspective: Flexibility is essential since security relationships must be revised whenever there are changes to: user tasks and hence their assigned user roles; user role requirements; the way an application is utilized; and the application’s software structure (i.e., deleting and/or modifying classes/methods). Often such changes are driven by the evolution of an application and are expected in its normal life cycle. Any enforcement mechanism must support all such changes with little or no impact to the application and environment in which it functions. Flexibility differs from extensibility since it involves changes to existing structure rather than just additions.

URBS Enforcement Mechanism Perspective: A flexible enforcement mechanism is required, since it is likely that as the application is developed and utilized, the security restrictions and/or the implementation of the enforcement mechanism may change. If changes required an entire new redefinition and implementation of the mechanism, and further required that the entire application be rebuilt, then the mechanism would not adhere to the foundations and claims of the object-oriented paradigm.

*Hiding & Encapsulation: “The less I know the less I need to worry about.”*

Application Perspective: The enforcement mechanism should be encapsulated and hidden from the other components and users of the application. A hidden enforcement mechanism will not increase the complexity of developing, understanding, extending, modifying, and/or maintaining the application software. In addition, the encapsulation of security information will help ensure against malicious access while also providing ease of use.

URBS Enforcement Mechanism Perspective: The internals of the mechanism should encapsulate the security data and hide the implementation in order to support the safe development of the enforcement mechanism. This supports representation independence, allowing the implementation of the security mechanism to change as long as its view (public interface) to the application is unaltered. This is critical for the extensibility and flexibility goals! Clearly the mechanism will need to be developed and defined in the same environment as any other object-oriented application. The unauthorized access of security information may also be considered to be extremely dangerous to the development and utilization of an application. This last point raises the question as to whether there needs to be an enforcement mechanism for the URBS enforcement mechanism!
• **Reusability:** "Is there a generic enforcement mechanism?"

Applic"..."tion Perspective: As new applications are built from existing classes or c...n application, new separate enforcement mechanisms should not have to be totally regenerated. The existing mechanism should be independent of the environment, or application, for which it was originally created. Any definitions that are still applicable, should be automatically reused in the new application. For example, assume that a particular UR was assigned access to a number of methods of an OT. As new applications are generated, using the previously developed OT, the enforcement mechanism should provide the same security access control for the same role in the new application for those methods that were reused from the original OT. This concept is essential if the reusability claims of object-oriented paradigm are to be attained.

URBS Enforcement Mechanism Perspective: As finer granularity levels of security are required or even new security definitions are generated, then the mechanism components should be reusable to provide such new capabilities readily. The degree that new code is required should therefore be minimal. In the object-oriented world, the use of generics for type-parameterizable structures like lists, sets, and so on, is widespread, promoting reuse and minimizing inconsistency. In the same manner, the attainment of a “generic” URBS enforcement mechanism, parameterizable by application specific...s into the object-oriented paradigm.

What, if any, guidance can be derived from the above goals regarding the approach or approaches to the design and development of a URBS enforcement mechanism? Is any mechanism capable of achieving these goals? There are a number of preliminary observations that can be made.

First, from the various goals it is realistic that the enforcement mechanism can be generated as a separate object-oriented application or component of an application. In either case, it seems evident that the mechanism must be developed under the object-oriented paradigm in order to attain the aforesaid goals. The security component must also be able to be elegantly attached to an object-oriented application. The attachment must be hidden as much as possible, shielding the user from its presence. This in turn will also hide the URBS enforcement mechanism. However, it is important to note that it will likely be impossible to hide all aspects of the URBS enforcement mechanism, particularly from the software engineers that are designing, developing, and maintaining an object-oriented application. This issue will be further discussed in Section 5.2.

### 5.2 Approaches for a URBS Enforcement Mechanism

In this section, three different approaches to generating code for a URBS enforcement mechanism are proposed, discussed, and critiqued. Each of the three approaches are discussed in the context of URBS as supported by the ADAM environment. Specifically, the focus is on the user-role definition hierarchy (see Figure 6.3 again), its user roles, and the assigned/prohibited methods for each user role. The *brute-force approach (BFA)* for an enforcement mechanism relies on code-level modifications to methods to support the security requirements given in the URDH, with little attention given to object-oriented precepts and principles. 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. Finally, the *URDH-class-library approach (UCLA)* also utilizes inheritance to implement the enforcement mechanism, but does so from the combined perspective of the application's URDH and OT-class library. These three approaches are presented under the assumption that there is no user code involved in the system, i.e., a static system. Users are not able to write programs to access data directly. Rather, users utilize tools that embody the apropos security code to enforce the required URBS policy. All three approaches are discussed in this section, with a concluding section dedicated to their comparison and contrast against the goals given in Section 5.1.
It should be noted that some of the approaches are based on capabilities that may not be supported in current object-oriented languages. For example, the ability to dynamically modify and extend compiled class libraries, without recompilation, is not a feature of C++. In addition, to assist in explaining each approach, C++ code segments are provided. C++ was chosen due to its popularity and the fact that ADAM generates C++ code. Ontos C++ was not chosen to avoid the explanation of Ontos-specific concepts. However, the transition of each example from C++ to persistent objects in Ontos is a straightforward exercise. Note that all three approaches have all been partially implemented in an attempt to verify/refute the ideas presented herein.

5.2.1 The Brute-Force Approach

In the brute-force approach, BFA, for an enforcement mechanism, code-level modifications to the methods of an OT are made that add control-flow statements based on URD/assigned method information. The approach can be understood from the following example from HCA:

```cpp
class Prescription {
public:
    Get_Prescription_No(....);  
    Set_Prescription_No(....);  
    Get_Pharmacist_Name(....);  
    Set_Pharmacist_Name(....);  
    Get_Medication(....);       
    Set_Medication(....);       

private:
    char* prescription_no, pharmacist_name, ..... 
};
```

User Role: Staff_RN

URD/Assigned Methods: Get_Prescription_No(....);  
Get_Pharmacl Name(....);  
Get_Medication(....);

In the example, Staff_RN has been assigned three methods. To allow this access, the following brute force code could be added to the Get_Prescription_No method, with corresponding changes to the other two methods. In the method, the validity of the UR is verified prior to the execution of the code that implements the method's functionality.

```cpp
char* Prescription::Get_Prescription_No (....) 
{
    // Check if a valid user is accessing the method
    if (User.User_Role() == Staff_RN) {
        // .... method code for the access ..... 
    }
    else
        return (NULL);
}
```

In the above code segment, the access to the Get_Prescription_No method is controlled by the "if" statement which verifies the current role of the user to be that of a Staff_RN. If other roles required access to the method, then additional "or" clauses (|| User.User_Role() == Manager) would be added. The OT User stores the role(s) of the current user, effectively implementing the assigned
methods from the URDH. A NULL is returned to the calling instance if the user role has not been assigned the Get_Prescription_No method in the URDH.

While the BFA is very simple to understand, it suffers from many drawbacks. The automatic generation of the "if" statements could be easily done as part of the code generated by ADAM. However, the approach doesn’t meet many of the goals of an ideal URBS enforcement mechanism. The mechanism is hidden from the outside world, since it is not part of the public interface. But its encapsulation is poor, since it is distributed across all of the methods of all of the OTs. That is, there is no easily identified enforcement mechanism for BFA that is encapsulated, making extensibility and flexibility impossible to attain. When modifications to user roles, methods, and so on, are made, then the code for the OTs would have to be regenerated by ADAM and recompiled. Such an approach is unfeasible for both large applications with many users or for applications whose roles are constantly changing. It should be noted, that for well-defined, non-dynamic applications, the approach may have merit.

5.2.2 The User-Role-Subclassing Approach

In the URDH, the assigned and prohibited methods for a user role correspond to the methods that it can and cannot invoke. The information in the URDH is organized on a role-by-role basis, and consequently, a given UR may be assigned/prohibited privileges on multiple OTs that span the application. Thus, the assigned and prohibited methods for a UR are partitionable on an OT-by-OT basis. If this exercise is performed for all URs in the URDH, then the result is that for each OT, we know exactly which URs access the OT, and further for each of the URs, which methods of the OT are assigned and/or prohibited. In the user-role-subclassing approach (URSA), each OT of the application has a group of subclasses, based on the different URs that have some subset of assigned and/or prohibited methods from the OT. 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). While this sounds similar to BFA, the key for URSA is to achieve this result via inheritance.

The implementation of an enforcement mechanism using URSA, unlike BFA, raises the abstraction level to the method level as given in the OT/class declaration. Any OT in an object-oriented application has a public interface composed of methods. If a user role, say R, was to inherit from an OT, it would inherit all of the methods from the public interface. But, in the URDH, it may be the case that R has been assigned to only a subset of the OT’s methods. The key for URSA is to “turn-off” methods from the OT that haven’t been assigned. To achieve this result, the C++ concepts for virtual functions and void return types are employed. These concepts are most easily explained using a subset of HCA, with the C++ class declaration and two URs given below:

```cpp
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, ..... 
};

UR: Attending_MD
PM: Set_Pharmacist_Name
AMs: All of the Rest

UR: Staff_RN
AMs: Get_Pharmacist_Name
AMs: Get_Prescription_no
```

25
Get_Medication
PMs: Set_Prescription_No
Set_Pharmacist_Name
Set_Medication

Using this information, two C++ classes would be defined, one for each UR, that both inherit from Prescription, as follows:

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

class Attending_MD_Prescription : public Prescription {
    public:
        virtual void Set_Pharmacist_Name(...) {return;};
};

Notice that in each subclass, each prohibited method is virtual, but its return type has been reset to void, allowing the subclass to effectively "turn-off" that method, i.e., when invoked, the method of the subclass executes and returns nothing. All methods for Prescription that haven't been "turned-off" are still accessible.

While this code is easy to generate, the reason things work at runtime rely on dispatching, substitutability, and mutability. Recall that in dispatching, when a method is invoked, the runtime environment looks at the type of the invoking instance to determine which method is actually called. In the previous class declarations, if the code:

    ptr->Set_Prescription_No();

is invoked, then the method that is actually called depends on the type of ptr. If ptr is of type Prescription*, then Set_Prescription_No as defined in Prescription is called. If ptr is of type Staff_RN_Prescription*, then Set_Prescription_No as defined in Staff_RN_Prescription is called, which results in no effect, since this is a prohibited method for Staff_RN. Substitutability allows a instance of type Staff_RN_Prescription to be used where a Prescription instance is expected, since the former is a subclass of the latter. Mutability supports the casting between subclass and superclass to achieve the aforementioned behavior at runtime. While a given user role's subclass invokes all of the methods from the superclass, the end-result is that the UR only has results returned (or actions executed) for those methods that were assigned in the URDH. Basically, in HCA, if the Set_Prescription_No is requested by a individual whose UR is Staff_RN, then the method associated with Set_Prescription_No of the Staff_RN_Prescription subclass is executed and no value is returned. If the access was made by an Attending_MD UR then the actual method would be executed to allow the prescription to be set.

There are two implementation options that we have examined. The first, which compiled and executed, contains code from the main() program:

    // Various include files - omitted for brevity
    main()
    {
    // Variables for the main program
        char user_name[20], user_role[20];
        Prescription* P;

    26
Staff_RN_Prescription* SP;
int Number;
char* Medication;

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

// Simulates a URDH and Authorization List
Current_User = new User(user_name, user_role);

P = new Prescription("Jessica", "3-9-95", 100, "Kitty", "Cold");

// This represents a piece of application code
if (strcmp(user_role, "Staff_RN") == 0)
{
    SP = new Staff_RN_Prescription("", ",", 0, ",", ",");
    SP->copy_object(P); //copy attributes from the parent
    SP->Set_Prescription_No(200); // Try to set prescription_no to 200
    Number=SP->Get_Prescription_No(); // Invokes valid method
    cout << "Number=" << Number << ",n"; // Prints 100
    P->copy_object(SP); //copy attributes from the child
    delete SP;
}

The copy_object function is defined on Prescription, to copy common private data from a parent to a child instance and vice-versa. This gives us the effect of substitution and mutation. The problem with this implementation option is that it requires the software engineer writing code to know a significant amount about security issues. While the code for the previously given class definitions can be generated by ADAM, the code in main cannot.

An ideal implementation option, would supply a degree of intelligence to the application code with respect to the possible user roles (from URDH) and the current role of the user. In this option, based on runtime casting from superclass to subclass, a Prescription pointer, p_rec would be utilized to reference all instances as the common parent. In the example below, a piece of application code is being written:

// Application Method to fill in a patient's personal information
void Fill_Patient_Prescription (......, Prescription* p_rec) {
    // Determine the role of the user currently
    switch (User.User_Role()) {
    case Staff_RN:
        ((Staff_RN_Prescription *)p_rec)->Set_Prescription_No(...);
        break;
    case Attending_MD:
        ((Attending_MD_Prescription *)p_rec)->Set_Prescription_No(...);
        break;
    }
    //...
}
With current C++ compiler technology, a switch statement would be needed to differentiate among the different URs. Once a match is found, a cast of \( p\_rec \) to the current role of the user is made. The extra set of parentheses around the cast of \( p\_rec \) is necessary to ensure that the object is first casted into the appropriate UR subclass prior to the \( \text{Set\_Prescription\_No} \) method being invoked. Since the object is casted first, the correct UR specific subclass method is invoked instead of the corresponding superclass method. While this solution has merit, it does require the use of switch statements. Any changes to the URDH or the application would require regeneration and recompilation, since switch cases requiring updating. In order to eliminate the switch statement, it would be necessary to change the runtime environment of the C++ compiler to take into account the UR of who's executing the code when determining which method to call. If such a change were to occur, then the single statement:

\[
p\_rec->\text{Set\_Prescription\_No}(\ldots);
\]

would replace the switch statement. Individuals designing and writing code would use the application OTs, with ADAM automatically including all relevant UR subclasses.

URSA offers many benefits over BFA. Clearly, it is more encapsulated, since the actions occur at the method level, with the code for UR subclasses easily generated by ADAM. Extensibility is possible, since one can add new subclasses for new URs to an OT (say \( \text{Prescription} \)), without affecting the other UR subclasses previously defined. In fact, the use of the virtual functions in the UR subclasses (say \( \text{Staff\_RN} \)) allows specialized varieties to be defined (say \( \text{IV\_Staff\_RN} \)), without impacting other portions of the inheritance hierarchy. But clearly, any new URs cause the application code to be changed, so that if clauses and/or switch statements can be updated. Flexibility is not as easily attained. If existing class and subclass declarations must be changed, it is possible to derive a new subclass of \( \text{Staff\_RN} \) to further restrict its capabilities by voiding out additional methods. Reuse is unclear; if changes to the C++ runtime environment are made, both the application and the mechanism should have a high degree of reuse.

Overall, in URSA, the subclasses refine the public interface with mutability exploited between \( \text{Prescription} \) and its UR subclasses to arrive at generated code that is relatively extensible. However, there are two unresolved issues to be addressed. First, in an actual implementation of HCA, \( \text{Prescription} \) instances are stored persistently. Whenever an individual (with a set of user roles) needs to access \( \text{Prescription} \), its loaded from the persistent store and muted to a form that matches the apropos UR. The reverse occurs to store a modified instance. The impact of persistent storage and retrieval must be examined to insure that URSA as proposed will work correctly. Second, the IDevEs role in URBS implementation must be considered. For example, in HCA, suppose that an IDevE is given access to a set of assigned methods to develop a patient/data/display tool. How much must the IDevE understand UR concepts in order to successfully implement a secure tool? Can code be automatically generated (in the form of a class library) to seamlessly handle the storage/mutation of instances based on URs? This and other questions are related to the many concerns raised earlier in Section 4.

5.2.3 The URDH Class Library Approach

When a URDH is created, both generalization and specialization are employed. Furthermore, inheritance is applied when passing assigned methods down through the URDH from UCs to UTs to URs. Since the security definition and privilege acquisition processes are so strongly tied to the URDH, it is logical to explore an enforcement mechanism for URBS that is bound to the URDH. In the \textit{URDH class library approach (UCLA)}, the class hierarchy of the application is unchanged, and a new class hierarchy is provided to support the enforcement mechanism. In this class hierarchy, each class represents a node of the URDH and the access rights of the node are specified via a set of validation methods. This is accomplished by first defining the access to all of the methods of the application to be false at the root class of the new class hierarchy. For each URDH node (UC/UT/UR), positive method access is defined based on the assigned methods that have been specified on the node. As the application executes, each method invocation must validate against the current UR.
Recall the URDH defined for HCA (see Figure 6.3 again). If we focus on a single UC/UT/UR path in the URDH, a new class hierarchy can be defined as follows:

\[
\text{URDH:}
\]

\[
\begin{align*}
\text{UC: Medical Staff} \\
\text{UT: Nurse} \\
\text{UR: Staff RN}
\end{align*}
\]

\[
\text{Assigned methods:}
\]

\[
\begin{align*}
\text{Get\_Prescription\_No}() \\
\text{Get\_Pharmacist\_Name}() \\
\text{Get\_Medication}()
\end{align*}
\]

\[
\text{URDH Class Library:}
\]

\[
\text{class Root: all Check\_* methods return False;}
\]

\[
\text{class Users: public Root}
\]

\[
\text{class Medical\_Staff: public Root}
\]

\[
\text{class Nurse: public Users, public Medical\_Staff {}
\]

\[
\begin{align*}
\text{int Check\_Prescription\_Get\_Medication}() \{ \text{return True;} \}
\end{align*}
\]

\[
};
\]

\[
\text{class Staff\_RN: public Nurse {}
\]

\[
\begin{align*}
\text{int Check\_Prescription\_Get\_Prescription\_No}() \{ \text{return True;} \}
\end{align*}
\]

\[
\text{int Check\_Prescription\_Get\_Pharmacist\_Name}() \{ \text{return True;} \}
\end{align*}
\]

\[
};
\]

In this new class library, the Root class includes new Check methods which are defined for all application methods from all OTs. Each of these Check methods return False at the Root level and will be turned on at lower levels (UC/UT/UR) by the assigned methods of the URDH. Assigned methods turn on the Check methods, as illustrated above by the UT Nurse and the UR Staff RN. The utilization of these Check methods will be illustrated shortly.

In addition to this new class library, there are modifications required to the application code that is generated by ADAM. First, a base class "Object" is defined for the application. The class Object has only one attribute, current\_user. To apply the URBS enforcement mechanism for UCLA, the software engineer only needs to inherit from the class Object, so that the validation of a method's invocation can be made based on the current user. A boolean function for checking user access will be inserted in each method defined in the application. In the example, some of the original application code must be modified to define and inherit Object:

\[
\text{class Object {}
\]

\[
\begin{align*}
\text{protected:}
\end{align*}
\]

\[
\begin{align*}
\text{Root\_* current\_user;}
\end{align*}
\]

\[
\begin{align*}
\text{...}
\end{align*}
\]

\[
};
\]

\[
\text{class Item: public Object{}
\]

\[
\begin{align*}
\text{...}
\end{align*}
\]

\[
};
\]

\[
\text{class Prescription: public Item {}
\]

\[
\begin{align*}
\text{public:}
\end{align*}
\]

\[
\begin{align*}
\text{Prescription(Root\_* u, char\_* Name, char\_* D, int No, char\_* Name1, char\_* Med);}
\end{align*}
\]

\[
\text{int Get\_Prescription\_No();}
\]
... 

```cpp
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;
    }
}

// Other Gets and Sets are similar and have been omitted for brevity
```

Notice that the code for each Get/Set method has been modified to utilize the current_user data field for invoking the Check methods which are defined in the previously given URDH class library. At runtime, a user must logon with his/her name and user role, creating a UR instance. The UR instance is passed to constructor of each application OT as instances are created (or loaded from a persistent store). The boolean function in each method will be checked whenever an application instance invokes a method. For the above code, when the current_user->Check... method is invoked, the runtime environment consults the URDH class library. If a True is returned, then the method is executed, since this signifies that the current user has a role that has been assigned the method that was invoked. The inclusion of the if statement in each method is similar to BFA, but doesn’t suffer from extensibility/flexibility problems since changes to the URs (in assigned/prohibited methods) occur in the URDH class library rather than the application class library.

To verify UCLA, we returned to HCA and implemented a simple access to Prescription by Staff_RN as follows:

```cpp
// Various include files - omitted for brevity
main()
{
    // Variables for the main program
    Prescription* P;
    char user_name[64];
    char user_role[64];
    int Number;
    Root* current_user;

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

    // Create and Simulate a new user
    if (strcmp(user_role, "Staff_RN") == 0)
    {
        current_user = new Staff_RN(user_name);
```
else current_user = new Root(user_name);

P = new Prescription(current_user,"Jim", "3-9-95", 1, "Ron", "Flu");
P->Set_Prescription_No(100);
Number=P->Get_Prescription_No();
cout << "Number=" << Number << \\
In this program segment, current_user is created based on the current user role. Since Staff_RN
doesn't have access to Set_Prescription_No, the boolean function:
Check_Prescription_Set_Prescription_No();
will return false and P->Set_Prescription_No(100) won't update the prescription number. Since
Staff_RN has access to Get_Prescription_No, the output of this method call will be Number=1.
The URDH class library approach offers many benefits. The approach resolves undo problems; since
no changes are made to actual attributes by unauthorized roles, there are no actions to be undone. The
approach appears to meet many of the goals of an ideal URBS enforcement mechanism, particularly
as related to extensibility and flexibility. As user roles are added, only the URDH class library must be
recompiled. Similarly, if a user role's security definition changes (changes to assigned/prohibited
methods), only the URDH classes would need to be modified and recreated. Like all approaches,
changes to the application will impact on both class libraries. The enforcement mechanism is somewhat
encapsulated, but clearly the need to inherit from Object does require the software engineer to know
something about security. On the whole, the UCLA approach hides the implementation of security
from the application and does make application code more streamlined than the two earlier approaches.
Finally, the automatic generation of the URDH classes, and the insertion of special execution code, can
be supported by ADAM.

5.2.4 Comparing/Contrasting the Three Approaches

In Tables 6.1 and 6.2, a comparison/contrast of the three different approaches against the four different
goals of Section 5.1 is given. Table 6.1 focuses on the perspective of the application; Table 6.2 on the
enforcement mechanism. In both tables, NO indicates that the approach doesn't support the goal, YES
signifies that it does, while PARTIAL provides the middle ground. In Table 6.1, UCLA has a slight

<table>
<thead>
<tr>
<th>Appr.</th>
<th>Extensible</th>
<th>Flexible</th>
<th>Hiding/Encap.</th>
<th>Reusable</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFA</td>
<td>NO</td>
<td>NO</td>
<td>YES/YES</td>
<td>NO</td>
</tr>
<tr>
<td>URSA</td>
<td>PARTIAL</td>
<td>PARTIAL</td>
<td>PARTIAL/YES</td>
<td>PARTIAL</td>
</tr>
<tr>
<td>UCLA</td>
<td>YES</td>
<td>PARTIAL</td>
<td>YES/YES</td>
<td>PARTIAL</td>
</tr>
</tbody>
</table>

Table 6.1. Application Perspective: Approaches vs. Goals.

advantage over URSA; that advantage is smaller in Table 6.2. The differences between both approaches
might disappear if the runtime environment for object-oriented programming languages was upgraded
to dynamically consider the user role when determining which method to execute. Clearly, in both
tables, BFA is a definite loser, lacking the key characteristics of extensibility, flexibility, and reuse.
Reuse is the most difficult to achieve in all approaches, since applications are very different and their
security requirements are very specific. Generic class libraries for security enforcement mechanisms offer
the best hope for achieving reuse. Overall, while BFA rated poorly against the four goals, URSA and
UCLA were comparable, especially when these two latter approaches are coupled with improvements
to the C++ runtime environment. URSA and UCLA seem versatile enough to allow different security
requirements to be realized on specific object types or on entire inheritance hierarchies, based on user
roles. Such enforcement mechanisms will allow software and tools to be made available that appear,
<table>
<thead>
<tr>
<th>Appr.</th>
<th>Extensible</th>
<th>Flexible</th>
<th>Hiding/Encap.</th>
<th>Reusable</th>
</tr>
</thead>
<tbody>
<tr>
<td>BFA</td>
<td>NO</td>
<td>NO</td>
<td>YES/NO</td>
<td>NO</td>
</tr>
<tr>
<td>URSA</td>
<td>YES</td>
<td>YES</td>
<td>NO/YES</td>
<td>PARTIAL</td>
</tr>
<tr>
<td>UCLA</td>
<td>YES</td>
<td>YES</td>
<td>YES/YES</td>
<td>PARTIAL</td>
</tr>
</tbody>
</table>

Table 6.2. URBS Enf. Mechanism Perspective: Approaches vs. Goals.

react, and behave in a customized fashion, based on the user role, thereby dynamically enforcing the security requirements.

6 Advanced Security Code Generation

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.

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 three approaches that use generics singularly, and in combination with exception handling to enforce URBS. In the first approach, generics are applied to URSA (see Section 5.2.2 again) to yield the generic URSA approach. A corresponding solution for UCLA is also possible, but has been omitted, since it is strongly similar in concepts and details. 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 BEA and GEA, 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.

6.1 The Generic User-Role Subclassing Approach

The generic user-role subclassing approach (GURSA) reformulates the techniques given in Section 5.2.2 by introducing a generic prescription class that encapsulates user-role specific access. As shown below, 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.

```cpp
#include "s_pres.h" // Staff_RN_Prescription Class from Section 6.2
#include "a_pres.h" // Attending_MD_Prescription Class from Section 6.2
```
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_Prescription_Name() { return(current_Ur->Get_Prescription_Name());}
    void Set_Prescription_Name(char* Name) { current_Ur->Set_Prescription_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);}
};

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 6.2 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 <stdio.h>
#include "item.h"       // Item from Figure 6.2.
#include "pres.h"       // Prescription from Figure 6.2.
#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;
    char user_name[64], char user_role[64], *Medication, *Name;
    int Number;

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, in order
to be able to correctly insert conditionals into the code.

These disadvantages led us to explore a second implementation alternative that replaces the condi-
tional with:

if (strcmp(user_role, "Staff_RW") == 0)
    Generic_Prescription<Staff_RW_Prescription> *PP
        = new Generic_Prescription<Staff_RW_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.

34
6.2 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 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_Photograph_name();          void Set_Photograph_name(char*);
  char* Get_Medication();               void Set_Medication(char*);
  int rtnt_int_check_valid_URI(int);    char* rtnt_str_check_valid_URI(char*);
  void set_int_check_valid_URI(int*,int); void set_str_check_valid_URI(char*,char*);
  void Check_URI();                     // Method to check if UR can access method.
  class Unauthorized_URI {}; // Exception handling methods follow.
};

There are six new methods that have been added. Check.URI is needed to verify that the current URI can invoke the desired method. For the purposes of this first example, we'll assume a simple table lookup. The class Unauthorized_URI 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 6.4. The remaining four methods are used to handle exceptions for the Prescription methods that return (rtnt) 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(rtnt_int_check_valid_URI(Prescription_No));
}

int Prescription::rtnt_int_check_valid_URI(int AInt)
{
  try { // try block has potential to raise exception
    this->Check_URI();
  }

  // catch block processes any raised exceptions
  catch (Prescription::Unauthorized_URI) {
    cout << "Attempt to access by unauthorized URI" << 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: MD, Date, Presc#, PharName, Medication
        P = new Prescription("Lois", "2-13-96", 1, "John", "Aspirin");
        Name = P->Get_Physician_Name();
        cout << "Name=" "\n";
        Date = P->Get_Date();
        cout << "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=" "\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 the 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_JR method as was previously discussed.

There are two advantages to BEA. First, the code for the rtIn, set, and Check_JR encapsulates the URBS code, hiding these details from the software engineer. Second, once the user role has been establish (via Current_User), the rest of the code given in main() doesn’t require the identification of UR. This is in contrast to the GURA code given in Section 6.1, 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 rtIn methods. Thus, reuse is not attained.

6.3 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.

```cpp
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_JR();
    class Unauthorized_JR { }; // Exception
    int rtIn_int_check_valid_JR(int);
    float rtIn_flt_check_valid_JR(float);
    char* rtIn_str_check_valid_JR(char*);
    char rtIn_chr_check_valid_JR(char);
    void set_int_check_valid_JR(int*, int);
    void set_flt_check_valid_JR(float*, float);
    void set_str_check_valid_JR(char*, char*);
    void set_chr_check_valid_JR(char, char);
    void prt_int_check_valid_JR(int);
    void prt_flt_check_valid_JR(float);
    void prt_str_check_valid_JR(char*);
    void prt_chr_check_valid_JR(char);
};
```

Notice that this class header contains rtIn and set methods for the four primary data types, along with the GS_Check_JR and Unauthorized_JR methods that are consistent with BEA in Section 6.2. 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; rtIns are similar to BEA with generic syntax overlaid, while sets simply have an assignment in place of the output statement in prt.

37
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.

    // Header file for Item
    #include "gen_security.C"
    class Item {
        // Private data has been omitted
    public:
        Item(char* Name, char* D); 
        char* Get_Protitioner_Name(); 
        void Set_Protitioner_Name(char*); 
        void Print_Protitioner_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

    38
};

// Header file for Prescription
#include "item.h"

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
};

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 SUC;
    else
        return FAIL;
}
The only difference between BEA and GEA is that the call to a prt (or rtn or set) method must be preaced 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 6.2.

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_Phalacist_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.

6.4 Advanced Exception Handling

The exception handling techniques discussed in Sections 6.2 and 6.3 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:

```cpp
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_SR();

    class Unauthorized_SR {
    public:
        int meth_id;
        Unauthorized_SR(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_SR exception

    class Logauthorized_SR {
    public:
        int meth_id;
        FILE *f_id;
        Logauthorized_SR(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.
        }
    }
```

40
}; // End Logauthorized UR exception

// rtn, set, prt methods as previously given without changes


template<class Type> void Gen_Security<Type>::GS_Check_UU()
{
    // Call Check of Owner Class
    if (Type::Check_UU(C_ID+M_ID) == FAIL)
        throw Unauthorized_UU(C_ID+M_ID);
    else
        throw Logauthorized_UU(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_UU)
    {
        cout << "Attempt to access by unauthorized UR" << endl;
        return; // return control without printing
    }
    catch (Gen_Security::Logauthorized_UU)
    {
        cout << "Logging of authorized access by UR" << endl;
    }

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

7 Looking Ahead and Further Readings

This chapter has contained an examination of URBS concepts and their attainment within the object-
oriented design model given in Chapter 5. To do so, the model was extended with security components
for node profiles (see Section 2 again). To illustrate the utility of URBS, both HCA and an example with
software development environments (see Section 3 again) were given. Given this background, a more
general discussion and exploration of security issues for the object-oriented paradigm was presented in
Section 4, followed by an application of many of these ideas to actual URBS approaches for security
enforcement using C++ in Section 5. These basic approaches were extended and reformulated to
advanced approaches in Section 6 using generics in combination with exception handling to have more
extensible and reusable secure code. To finish the chapter, the proposal of a change to the unit of
abstraction of the object-oriented paradigm from an OT to an inheritance hierarchy was examined.
The interested reader is referred to a large body of our efforts in user-role based security for object-
oriented models that has appeared elsewhere in the literature [8, 9, 10, 11, 12, 13, 14, 20, 21, 22].

One of the key concerns for this chapter is to attain URBS for object-oriented systems in a seamless
fashion that: facilitates the definition of URBS during application design; includes URBS as part of the
implementation via automatic code generation of generic, reusable security libraries, insulating software
engineers from security details; and, enforces URBS during runtime by customizing the behavior of
executing software based on role and therefore controls access to information. Given these efforts,
there are a number of issues that must be addressed, including:

- The inability to utilize generics as desired in Section 6.1 raises again the need to open a dialog
  between researchers and practitioners in security, programming language, and compiler fields.
Improvements and enhancements 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.

- URBS for object-oriented systems must be expanded to consider other languages like Ontos C++, Ada®®, 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 Ada®®, 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.

Progress on these two issues is crucial if we expect to successfully integrate URBS with object-oriented design and implementation.

NEED LOOKING AHEAD AND FURTHER READINGS....

Chapter 6 Exercises

1. To explore concepts related to user-role based security definition, develop a URDH for HTSS. As part of this development, examine the privilege definition process as related to assigned/prohibited methods and consistency criteria. Be sure to illustrate the iterative and/or incremental nature of the design process by providing different snapshots of the URDH at different time points. To assist you in the formulation of your answer for this question, you may need to define a set of OTs for an Item hierarchy.

2. In defining privileges with the URDH, three abstraction levels have been presented, based on user classes, types, and roles. Examples have shown that these levels can be utilized in different ways. For instance, in HCA, the UT Nurse has URs StaffRN, Manager, etc. This characterization is clearly geared towards job responsibilities. In the SDE example, the UT SE-Regular had Debug, Develop, Test, and Document URs. This versatility in the URDH is the motivating factor for not allowing sub-roles of URs (and so on), thereby simplifying the hierarchy by minimizing its depth. Is this a reasonable view? If so, provide additional explanation to bolster the argument. If not, why not, and more importantly, postulate the number of abstraction levels that should be provided.

3. In this exercise, you must further explore the ideas and issues related to the unit of abstraction for the object-oriented paradigm in general, and its impact on the different concepts and topics that have been discussed in Chapters 1 to 6. The basic assumption is: Suppose that instead of single OTs/classes, the unit of abstraction was an entire inheritance hierarchy. Your answers should address the impact (either positively or negatively) of the assumption.

4. Discuss the impact of changing the unit of abstraction to an inheritance hierarchy on basic object-oriented concepts and features, namely, the public interface and the private implementation (thereby including encapsulation and hiding). To assist in formulating your answer, the following questions might be useful:
   - In HTSS, can a descendant of Item access and change its private data directly? Why or why not?
   - Is the public interface defined for the entire hierarchy as a combination of all public methods? If so, what are the implications of such an approach? If not, why not?
• Do actions occur at only leaf nodes?

5. A more critical concern is the impact of revising the unit of abstraction on extensibility, namely, software reuse and evolution. Are reuse and evolution impacted by a change in the unit of abstraction? If so, how? If not, why not? Note: Your answer in part (a) might assist you in formulating your answer to this part of the question.

6. Generics are important to object-oriented development, since they provide a means for a IE to develop a single piece of code that can be customized based on type, thereby meeting multiple needs with a single effort. The majority of our discussion has considered a generic for a single OT/class. Does the concept of a generic inheritance hierarchy make sense? Why? If so, provide a generic hierarchy and indicate its expected utility. If not, make sure that you have a strong argument with supporting reasons. Note: Again, answers to parts (a) and (b) may be helpful.

7. Recall the reference model as presented in Section 6 of Chapter 2 and in ZM (ppgs. 13-23). Two features of the reference model that we discussed were: persistence by reachability and versions. Are these two features impacted by the inheritance hierarchy as a unit of abstraction? If so, how? If not, why not?

8. In Chapter 4, the ECRC approach for high-level design was presented as an abstraction and conceptualization technique to transition from the specification to a detailed design. Which, if any, of the characteristics of individual decks and/or hands and/or cards must be changed and/or modified to support an inheritance hierarchy as a unit of abstraction? Do any of these changes have an impact on the specification process as discussed in Chapter 3? Again, make sure that you provide a cohesive argument in your answer.

9. One of the other strengths of the object-oriented paradigm is its support for the testing process. The premise is that an object type can be designed, implemented, and tested in a manner that is independent of other portions of an application, and in a fashion that is more comprehensive. The claims are that the resulting implementation is more robust and less prone to maintenance. If you accept the premise/claims, the next question is on the impact of a change of a unit of abstraction on testing. Is there an impact? If so, is the testing process enhanced or degraded? If not, why not?

10. What impact does a change in the unit of abstraction have on the object-oriented design model as defined in Chapter 5? From a general perspective, examine the issue with respect to the profile concept. From a specific perspective, examine the impact on OTs, inheritance, RTs, and groups (Sections 4, 5, 6 and 7, respectively, of Chapter 5).

11. In Chapter 6, when assigning/prohibiting privileges to the different nodes of a URDH, the potential public interface for individual OTs was utilized. If the unit of abstraction is accepted, the impact on the privilege-definition process must be considered. Explore this issue.

References


Medical Staff: Collectively, responsible for all aspects of direct patient care.

Support Staff: Different support personnel that address non-medical needs of patients and maintain the physical building.

Other: Other individuals that have the potential to access limited portions of the health care database.

Nurse: Direct involvement with patient care on a daily basis.

Physician: Handle the medical needs (diagnosis, treatment, etc.) for patients.

Pharmacist: Control the supply and distribution of all drugs throughout the hospital.

Technician: Provide a variety of medical testing support for Patients.

Therapist: Evaluate patients and develop treatment plans for therapy.

Staff RN: Administer direct care to patients and implement the physician treatment plan.

Discharge Plng: Link between patients and outside agencies for care after discharge.

Education: Educate both the nursing staff and patients regarding new treatments and self care.

Manager: Responsible for the day-to-day operation of a nursing unit.

Director: (For Physician or Pharmacist) Responsible for the day-to-day operation of their respective department/medical service.

Private: The physician within his/her office/private-practice setting.

Attending: A physician that has privileges to admit and treat patients at a hospital.

Staff: Responsible for filling prescription orders for patients and analyzing appropriateness of drugs and dosages.

Lab: Perform/collection different tests involving body/blood on patients.

Radiology: Perform radiology based tests/treatments on patients.

Pharmacy: Distribute drugs to specific patients at correct times.

Physical: Perform physical therapy on patients at prescribed times.

Respiratory: Perform respiratory therapy on patients at prescribed times.

Occupational: Perform therapy geared towards returning the patient to the independent activities of daily living.

Support: Limited contact with patients on a day-to-day basis.

Prepare Room: Clean and prepare room after a patient is discharged.

Volunteer: Satisfy the needs and interests of patients by offering activities, reading materials, etc.

Security: Involved when prisoners/VIPs must be guarded/protected.

Figure 6.4. Sample Node Descriptions for HCA.
Staff RN: All clinical information for the patients that they are responsible for (referred to subsequently as clinical info.). Can write/modify a substantial portion of clinical information to record the results/patient progress. Cannot change a Physician’s orders on a patient.

Discharge Pmg: All clinical info. for patients. In addition, financial information is consulted, since patients might be placed in a continuing care facility or may require home visits from various health care professionals. Don’t have as much write access to clinical info. of a patient as Staff RN, but can write notes. Cannot change a Physician’s orders on a patient.

Education: More limited access to clinical data than Staff RN, but since they do teach patients after-discharge care (e.g., diabetic care, etc.), they do need access to a patient’s history. Like Discharge Pmg, can write notes which document a patient’s progress. Cannot change a Physician’s orders on a patient.

Manager: All clinical info. plus information required to transfer patients between units, information on the nurses that work in their unit (including shifts, staffing, and skill levels), and budgetary data. Write privileges of Staff RN plus extra privileges to read information on other units (censuses) and write summary and employee data on their own units. Cannot change a Physician’s orders on a patient.

Director: (For Physician) Information on physicians in their departments, budgetary data, clinical information on patients including summary data on trends and volumes. Write ability on employee data.

Private: All clinical info. on patients and who they can contact (nurses, technicians, therapists, etc.) regarding their patients. Also, insurance related data and other office-based data to maintain their private practice. Overlap of write privileges with Staff RN, but can also modify portions of clinical info. that issue orders.

Attending: All clinical info. on their patients. Similar to Private.

Director: (For Pharmacy) Information on pharmacists and technicians that are employed, budgetary data, summary information on drug distribution and usage, limited clinical info. on patients. Write ability on employee data.

Staff: Access to the Formulary database, all clinical info. on patients due to possible drug interactions, and prescription records.

Lab: Limited access to clinical info. on patients. They need to know what tests are required for which patients and when they are to be performed. Limited write access on clinical info. to record test results of patients.

Radiology: Similar to Lab user role.

Pharmacy: Similar to Lab user role. May need access to Formulary database.

Physical: Access to clinical info. that is greater than Lab user role but less than Staff RN. Can write notes on patient’s progress which are permanently recorded into the medical record.

Respiratory: Similar to Physical user role.

Occupational: Similar to Physical user role.

Prepare Room: Very limited clinical info. on patients - discharge date and time. No write access is allowed.

Volunteer: Very limited clinical info. on patients - names, location, restrictions (food/smoking), and interests. No write access is allowed.

Security: Very limited clinical info. on patients - duration/location of prisoners/VIPs. No write access is allowed.

Figure 6.5. Sample Role-Security Requirements for HCA.
<table>
<thead>
<tr>
<th>User Role:</th>
<th>Staff_RN</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Type:</td>
<td>Nurse</td>
</tr>
<tr>
<td>User Class:</td>
<td>Medical_Staff</td>
</tr>
<tr>
<td>Description:</td>
<td>Administer direct care to patients and</td>
</tr>
<tr>
<td></td>
<td>implement the treatment plan.</td>
</tr>
<tr>
<td>Security Reqr.:</td>
<td>All clinical information for the</td>
</tr>
<tr>
<td></td>
<td>patients that they are responsible</td>
</tr>
<tr>
<td></td>
<td>for. Can write/modify a substantial</td>
</tr>
<tr>
<td></td>
<td>portion of clinical information</td>
</tr>
<tr>
<td></td>
<td>to record the results/patient progress.</td>
</tr>
<tr>
<td>Asgd. Methods:</td>
<td>PPI2, M3.1, M3.3, M3.5, M4.2, M4.5, M4.7</td>
</tr>
<tr>
<td>Pro. Methods:</td>
<td>M2.6, M3.6, M6.10, M6.11, M6.12</td>
</tr>
<tr>
<td>Sub. Criteria:</td>
<td>as given previously</td>
</tr>
</tbody>
</table>

Figure 6.6. A Complete Node Profile for Staff_RN UR.

Figure 6.7. Initial Information for the User Role Staff_RN.

Figure 6.8. Selection of Assigned Methods for the User Role Staff_RN.
Figure 6.9. Conflict Identification Message.
Figure 6.10. Consistency Criteria Checking Message.
Figure 6.11. Sample Node Profiles for HCA.
Figure 6.12. A Sample SDE Application.

Figure 6.13. An Example of a User-Role Definition Hierarchy for SDEs.
Figure 6.14. Method Assignment in the User-Role Definition Hierarchy for SDEs.

Figure 6.15. Revised Method Assignments for SDE URDH.