TASK

A Framework for Collaborative Workspaces in Java

by

Kenneth Earle Hussey

BCSH, Acadia University, 1997

Thesis
submitted in partial fulfillment of the requirements for
the degree of Master of Science (Computer Science)

Acadia University
Fall Convocation 1999

0-612-45370-7

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Several attempts have been made in recent years to develop a useful environment to support collaborative work. The success or failure of these environments can perhaps be attributed in part to the degree to which they support a small number of desirable features. The purpose of this thesis is to describe the conceptual model, design specification, and implementation of a framework that, through its support for these features, facilitates the development of collaborative workspaces in Java.

GLOSSARY

action. An operation that can be directly invoked by actors in TASK

activation. The execution of a computational or algorithmic procedure

actor. An individual who collaborates, or coacts, within TASK; an entity outside a system that interacts with use cases

aggregation. A form of association that represents a whole-part relationship between an aggregate and its component part(s)

API. Application Programming Interface

assertion. A statement that should always be true and can only be false in the event of an error

association. A conceptual relationship between classes in which each class plays a distinct role and for which each role has its own multiplicity; a relationship that describes a set of semantic connections among tuples of objects

association class. A modeling element that has both association and class properties

association role. The end of an association where it connects to a class

asynchronous interaction. An interaction in which actors are not simultaneously involved in an action

attribute. A property possessed by a class

behavioral pattern. A design pattern that deals with communication between objects and classes

CGI. Common Gateway Interface

class. A description of all objects with similar structure, behavior, and relationships

class pattern. A design pattern that is concerned with static relationships (associations and subtypes) between classes

coaction. A synonym for collaboration

collection. A single object representing a group of other objects, referred to as elements of the collection

constraint. A restriction on one or more values of (part of) a model or system

context. The scope within which an actor is engaged in actions at a given point in time

CORBA. Common Object Request Broker Architecture

CPU. Central Processing Unit

creational pattern. A design pattern that abstracts the instantiation process

DCOM. Distributed Component Object Model

design pattern. A description of communicating objects and classes customized to solve a general design problem in a specific context

direct interaction. An interaction between actors in which an intermediate object is not involved

down-calling. A technique for ensuring that both the syntax and the semantics of an interface are consistent

focus. The collection of scopes with which an actor is associated

generalization. An inheritance relationship between a more general element and a more specific element

groupware. Computer systems designed to support groups of people working towards a common goal

GUI. Graphical User Interface

HTTP. Hypertext Transfer Protocol

IIOP. Internet Inter-ORB Protocol

implementation method. A method that is restricted to the inheritance hierarchy but can be overridden to provide the implementation for an interface method

indirect interaction. An interaction between actors in which an intermediate object is involved

interface. A collection of operations used to specify a service provided by a class or component

interface method. A publicly accessible method that can be directly invoked but cannot be overridden

invariant. A restriction on a class, type, or interface that specifies conditions which must hold true for all instances of that class, type, or interface

JDK. Java Development Kit

key. A mechanism of limiting the behaviors of actors within TASK

LAN. Local Area Network

lifeline. A dashed vertical line representing an object in a sequence diagram

message. A communication between objects which results in some activity

MUD. Multi-User Dungeon

object. An instance of a class with a well-defined identity

object pattern. A design pattern that involves dynamic relationships, which can be changed at run-time

object serialization. A technique in Java that converts data structures into a common data stream that is independent of processor or operating system

OCL. Object Constraint Language

OMT. Object Modeling Technique

one-to-many interaction. An interaction between one actor and many other objects

one-to-one interaction. An interaction between one actor and one other object

OOSE. Object-Oriented Software Engineering

operation. A process that a class knows how to carry out

ORB. Object Request Broker

PC. Personal Computer

persistence. The ability of an object to save its state so that it can be restored and used at a later time

postcondition. A condition that must be true immediately after the execution of an operation

precondition. A condition that must be true before an operation can be executed

purpose. What a design pattern does

realization. A semantic relationship between an interface and a class that realizes or implements it

RMI. Remote Method Invocation

scope. A frame of reference for the actions that actors engage in as they collaborate in TASK; whether a design pattern applies to classes or objects

structural pattern. A design pattern that deals with the ways in which classes and objects are combined to form larger structures

subtype. A generalization relationship in which an instance of the subtype (or child class) is also, by definition, and instance of the supertype (or parent class)

synchronous interaction. An interaction in which actors are simultaneously involved in an action

TASK. Tools, Actors, Scopes, and Keys

tool. A means by which actions are performed in TASK

UML. Unified Modeling Language

URL. Uniform Resource Locator

use case. A sequence of actions performed by a system that yields an observable result to an actor

ACKNOWLEDGMENTS

I wish to thank the following people, without whom this thesis would not have been possible:

- my thesis supervisor, Tomasz, for his patience and understanding
- my family, for their love, encouragement, and support
- my wife, Angela, for believing in me even when I didn't
- the little train that could, for inspiring me to think I can

1 INTRODUCTION

Computers have been used in educational settings for many years, but until recently, they have been used mostly in labs rather than in the classroom. Over the past few years, several universities have introduced the concept of electronic classrooms, in which each student has access to a networked computer. Some institutions have also experimented with "studio" classrooms, where teams of students sit around a table and collaborate to solve problems given by the instructor. The instructor is often supported by teaching assistants who supervise students by physically moving around the classroom and joining student teams. Providing these facilities is a very expensive initiative, and is only feasible if it is cost-effective. In an educational setting, this translates to increased efficiency of learning.

Perhaps the best learning processes are those that emphasize interaction and teamwork. Computer systems designed to support groups of people working towards a common goal are referred to as *groupware*. Unfortunately, in existing electronic classrooms, typically the only form of computer-based communication is through the use of standard Internet facilities, such as Web browsers or specialized client applications. While these systems may prove to be useful for the overall coordination and administration of courses, they will ultimately be found lacking.

These issues affect not only educational institutions but any organization that can benefit from the use of groupware, such as a software development team. Software development is a process that consists of a well-defined set of activities involving roles played by actors with a common goal. Formality of, and adherence to, the structure of these activities varies, but in general the process is iterative and incremental, and typically involves specification, analysis, design, implementation, testing, and evaluation. Quality software development results from successful modeling, monitoring, and managing of these activities. Several attempts have been made to provide automated support for this endeavor, but with only limited success. In particular, existing tools lack explicit support for teamwork across geographical boundaries, formal and informal communication, team cooperation and coordination, problem reporting and resolution, and quality assurance.

Much success using the LambdaMOO environment (see [Curtis 1993], [Curtis & Nichols 1993], and [Evard 1993]), among others, indicates that developing a framework for creating useful collaborative environments is possible. The base functionality provided by LambdaMOO is relatively minimal. Its utility results from its extensibility. As a framework for software development, however, it does still suffer from the legacy of spatial metaphors. What is needed is a fully extensible environment that is similar to LambdaMOO, but which relaxes the concept of space. Such a framework would provide a more convenient forum for communication and development, and give more freedom to developers to do what they do best - develop software.

Too much effort is being spent by researchers to develop environments from the client or user interface perspective. What will be achieved by these activities is merely a more convenient interface to features that could have been provided by an environment like LambdaMOO in the first place. This is not acceptable. We must change our focus to address the fundamental ways in which we communicate and interact while acting within the roles we play, in order to develop a framework that can be later complemented with a useful client interface which will (and should) be modified by users of the system anyway.

Virtual environments such as Multi-User Dungeons (MUDs) change a user's sense of orientation, time, presence, awareness, movement, and actions. Spatial metaphors are suitable (and even useful) in a role-playing environment, and indeed this was the target domain of the first MUD. The notion of space is perhaps too restrictive within the context of software engineering, or any social work environment for that matter. In the past, collaborative virtual environments have been designed to mimic the notion of space, hoping to improve usability based on users' familiarity with the physical nature of the real world. However, these environments have missed out on one of the most useful aspects of virtual environments, where spatial limitations are simply not necessary. Conversely, the notion of social context is a necessity in these environments. What is needed is a relaxation of the traditional views of virtual space, to give way to a metaphor which encompasses both spatial (to a limited extent) and social aspects.

An illustrative example of the inadequacy of the spatial metaphor is the use of exits that represent the links between rooms in traditional collaborative environments. In such environments, the exit construct introduced a logical separation between spaces and provided a closer model of the physical world than rooms could alone. However, after using such an environment for some time (and discovering its basic topology), the concept of an exit becomes more of a hindrance than a help. Indeed, most users make much greater use of a `teleport` (or equivalent) command, if one is available, to jump from room to room, essentially eliminating the need for exits. This phenomenon in turn leads to a more important consequence, the disintegration of the spatial metaphor itself.

Furthermore, focusing on specific artifacts to be manipulated by users is a waste of time for the designers of collaborative virtual environments. Attempts to provide a set of tools with the aim to solve all development problems will result only in failure. This is evident in the myriad of research projects in the past that tried to produce the miracle application. What is clear, however, is that an environment needs to support certain features to be useful in collaborative work. They include the following:

- *Frames of reference for collaborative activities.* Traditional environments attempted to provide this with the notion of rooms, but this is too simple a model. The frames of reference that we find ourselves being involved with stem not from our physical location, but rather from the many roles we play within (and without) an organization. At any one time, we may focus on the context surrounding a particular role, with its distinct social (and cultural) meaning. In reality, however, we remain responsible for all of our roles, implying that we need to be part of more than one frame of reference concurrently. This is not possible using a spatial model, which limits our participation in activities on a per-room basis.
- *A means of communicating within and between these frames of reference.* This is typically done in MUDs by representing users as objects, or players, in the environment and providing commands to allow these objects to interact. Various forms of communication exist, such as synchronous vs. asynchronous, public vs. private, coordinated vs. uncoordinated, one-to-one vs. one-to-many, etc.. Another important issue is awareness. In an environment that facilitates multiple frames of reference containing multiple users, a mechanism to provide feedback on the attentiveness of users within the environment is indispensable. Several attempts have been made to provide this in traditional environments, but with only limited success.

- *Tools to support the activities we perform within these frames of reference.* The exact nature of these tools depends on specific activities and cannot be wholly determined at design time. This is an issue overlooked by many systems designers. In order for an environment to be useful, it must be both general and specific at the same time, but more importantly, it must be extensible and adaptable. Deciding which tools are needed, and indeed which will be used, before an environment is deployed only makes the environment more rigid. A structure which allows the integration of new and existing tools into the environment, however, is quite useful (but difficult to achieve).

Building on concepts developed in previous research (see [Hussey 1996] and [Hussey & Tomek 1996]), I introduce a general framework to support collaborative workspaces, called TASK, which stands for Tools, Actors, Scopes, and Keys. Like LambdaMOO and other existing collaborative virtual environments, TASK is an extensible environment that facilitates real-time communication between its users. In contrast to LambdaMOO, however, communication in TASK is not text-based, but instead utilizes an event notification mechanism that allows users to share and interpret information about the activities that take place as a result of collaboration. Rather than attempting to provide a complete set of applications to solve a specific problem, as some of its predecessors did, TASK aims to be a general foundation that can be extended with tools to suit any target domain.

This thesis will describe in detail the conceptual model, design specification, and implementation of the TASK framework in an effort to demonstrate how it supports these features, and in turn facilitates the development of useful environments for collaborative work. More specifically, analysis of the conceptual model and its associated constraints (using the Object Constraint Language), specification of the design (using the Unified Modeling Language) and the application of patterns (Mediator vs. Observer vs. Event Notifier), and implementation of collections, assertions, remote objects (Remote Method Invocation vs. Voyager), and object serialization in Java will be discussed. It is assumed that the reader has a basic working knowledge of Java and the object-oriented paradigm.

2 CONCEPTUAL MODEL

In this chapter, I present a specific example of a studio classroom at Acadia University, which I believe applies equally well to any collaborative working environment [Hussey & Müldner 1998]. Consider a classroom in which the instructor and all participating students have access to networked computers and can interact, for example, through the use of downloadable files, an e-mail facility, and a projector which can display images from a single computer. The instructor is a member of the Faculty of Computer Science, and the students are undergraduates in the Bachelor of Computer Science program and enrolled in the introductory programming course, COMP 1013. The students are organized into groups of four to facilitate discussion of problems posed by the instructor. One expects that the following hypothetical interactions would be possible:

1. *Instructor to students:* The instructor gives a verbal introduction to the problem to be solved during the day's class.
2. *Instructor to students:* The instructor makes a file required to solve the problem available for the students to read.
3. *Student to students:* A student discusses the problem proposed by the instructor within his or her group of four students.
4. *Student to student:* A student sends an e-mail message to a student in another group asking if she or he can clarify some aspect of the problem.

5. *Instructor to student:* The instructor gives a student exclusive access to the projector, perhaps revoking other students' access to the projector.

6. *Student to instructor and students:* A student shares her or his group's solution to the problem with the rest of the class by displaying it on the projector.

2.1 Objects in TASK

Designing a framework to model a specific problem is somewhat similar to designing an object-oriented computer program; it consists of identifying the underlying objects and interactions between these objects. The collection of all objects relevant to a given problem might be called a problem's domain. In our electronic classroom, the domain consists of objects such as the following:

- students and instructors
- faculties, programs, classrooms, and groups of students
- files, e-mail facilities, and projectors
- objects providing access restrictions

In modeling interactions between these objects, it is useful to consider different kinds of objects. The TASK framework consists of four basic kinds of objects - actors, scopes, tools, and keys. In the following sections, I briefly describe each of these concepts, and how they can be used as a model of our case study. Note that TASK is a virtual setting, defined by the learning activity rather than by the boundaries of a physical classroom.

2.1.1 Actors

Collaboration occurs when two or more individuals engage in actions that are directed toward a common task (hence TASK). As such, I use the term ***actor*** to refer to the individuals who collaborate, or coact, within TASK (*coaction* is actually a synonym for collaboration). An actor, then, is the embodiment of a physical user and represents an entity involved in performing an action, for example a student, an instructor, or a programmer. I will refer to an actor using the impersonal "it", rather than she or he.

In our electronic classroom, there are $n + 1$ actors: n students and one instructor. The user embodied by each actor has access to a computer, and the computers are networked. The resulting interactions between actors are *direct* if no intermediate object is involved. Interactions 1 and 3 above are examples of direct interactions. *Indirect* interactions, on the other hand, involve a mediator object, such as a projector or an e-mail facility. Interactions 2, 4, 5, and 6 are examples of indirect interactions.

In addition to being direct or indirect, interactions between actors can occur in one of two modes. *Synchronous* interactions are those in which actors are simultaneously involved in an action. Interactions 1, 3, and 6 above are examples of synchronous interactions. Typically, synchronous interactions involve real-time communication in a manner similar to using a talk or chat program. This real-time aspect is the key distinction between TASK (among other virtual collaborative environments) and currently available groupware facilities. *Asynchronous* interactions are those in which actors are not simultaneously involved in an action. Interactions 2, 4, and 5 are examples of asynchronous interactions. Asynchronous communication is especially useful in a groupware application for situations where users are separated by a (geographical) time difference.

Interactions can also be categorized by cardinality. The two most common cardinalities in TASK are *one-to-one* and *one-to-many*, although others are possible. Interactions 4 and 5 above are examples of one-to-one cardinality; interactions 1, 2, 3, and 6 are one-to-many.

2.1.2 Scopes

Another important aspect of collaboration is the context within which it takes place. Indeed, our actions have little cohesive meaning unless they occur within some sort of boundary or frame of reference. This frame of reference is determined not merely by physical orientation, but also (more importantly) by the role we play in the collaboration. In our everyday lives, we may be responsible for many different roles, but at any one time we concentrate on the context surrounding a particular role, with its distinct social (and cultural) meaning.

A ***scope*** is a frame of reference for the actions that actors engage in as they collaborate in TASK. An actor enters a scope when it takes on a particular role associated with, or engages in some action bounded by, that scope. Conversely, an actor exits a scope when it is no longer interested in the actions that take place there. As a result, scopes are dynamic rather than static in nature - as actors enter and exit a scope, it binds, or holds a set of actors interacting within its boundary. In other words, a scope object consists of a number of actor objects (and other objects introduced later); when actor A enters a scope S, A is added to S, and when it exits S, A is removed from S.

Several examples of scopes exist within our electronic classroom case study. Acadia University represents the scope of faculties and programs that form the unique environment commonly associated with an institution of higher learning. The Faculty of Computer Science is the scope of all professors that research and teach computer science at the university, while

Bachelor of Computer Science is the scope of students that study computer science there (for the sake of simplicity we ignore specializations of this program). COMP 1013 is also a scope; it binds the instructor and student actors as they engage in the learning activities associated with this course. Finally, the groups in which the students are arranged as part of the studio classroom format are also examples of scopes.

Just as we may play a number of different roles in everyday life, so too can an actor be a member of more than one scope; in other words, an actor is associated with a collection of scopes. Borrowing terminology from awareness theory (see [Rodden 1996]), this collection of scopes can be called the actor's *focus*. In our case study, the instructor's focus consists of Acadia University, Faculty of Computer Science, and COMP 1013. The focus of a student consists of Acadia University, Bachelor of Computer Science, COMP 1013, and the group to which the student belongs. An actor's focus changes every time it enters or exits a scope; for example, when a student moves from one group to another. Note that an actor's focus consists of all scopes that it may be interested in, but the actor concentrates on only one of them at any given time. I call this specific scope, within which the actor is engaged in actions at a given point in time, the actor's *context*. In our case study, the instructor's context is COMP 1013, and a student's context is the group to which the student belongs. As with its focus, an actor's context changes every time it concentrates on a different scope; for example, the instructor who moves on to teach another course will have a different context.

It is perhaps intuitive that scopes be nested to form a sort of hierarchy. A useful metaphor for this is a file system of directories and files, in which directories contain files but can also contain other directories. In the same sense, scopes can form the context for actors engaged in actions but can also hold other scopes. Hence, a scope may consist not only of actors, but also other scopes (and other objects described below). In the case of nested scopes, before entering a particular scope, an actor must first enter the scope in which it is held.

The hierarchical relationship between the scopes in our electronic classroom case study is shown below. In particular, the Acadia University scope holds the instructor and student actors, and the Faculty of Computer Science, Bachelor of Computer Science, and COMP 1013 scopes; the Faculty of Computer Science scope holds the instructor actor; the Bachelor of Computer Science scope holds the student actors; the COMP 1013 scope holds the instructor and student actors, and the group scopes; the group scopes each hold four student actors. The focus and context of each actor can also be clearly seen below. For example, while the instructor actor appears under the Acadia University, Faculty of Computer Science, and COMP 1013 scopes (together comprising its focus), it is concentrating on the COMP 1013 scope (its context), under which it appears in bold typeface.

- **Acadia University**
  - *Instructor*
  - *Student 1*
  - ...
  - *Student n*
  - **Faculty of Computer Science**
    - *Instructor*
  - **Bachelor of Computer Science**
    - *Student 1*
    - ...
    - *Student n*
  - **COMP 1013**
    - **Instructor**
    - *Student 1*
    - ...
    - *Student n*
    - **Group 1**
      - **Student 1**
      - **...**
      - **Student 4**
    - **...**
    - **Group n/4**
      - **Student n-3**
      - **...**
      - **Student n**

2.1.3 Tools

In describing interactions between actors, it is useful to introduce the concept of a ***tool***. Tools are the primary means by which actions are performed in TASK and, as with actors, they are bounded by scopes. This has two consequences. First, the actions provided by a tool can only be performed by actors bounded by the scope in which it is held. Second, only the actors held by this scope are aware of the results of the performed action(s). Another aspect of tools is that they can be taken or dropped by actors, thus providing a means for their displacement between scopes. That is, an actor can take a tool from one scope, change its context, and then drop the tool in another scope. An actor that has taken a tool becomes a frame of reference for the tool until it is dropped. In this sense, an actor represents a kind of private scope for tools in TASK.

Within our electronic classroom, the files, e-mail facility, and projector represent examples of tools. The files are bounded by the COMP 1013 scope, and might provide actions to read (as in interaction 2 above), write to, and execute the file. The e-mail tool is bounded by the Acadia University scope, and might provide actions to check for, or send (as in interaction 4 above), new messages, and to read, reply to, or forward existing messages. The projector tool is bounded by the COMP 1013 scope, and might provide actions to turn on, or turn off, the projector, and to display an image from a single computer (as in interaction 6 above).

Although tools are the primary means of performing actions, other objects in TASK can provide actions as well. For example, actions exist to create and destroy every kind of object within the framework, and scopes provide actions to enter and exit their frames of reference (as alluded to earlier). However, a complete set of actions required by actors in TASK cannot be wholly determined at design time. In order for a virtual collaborative work environment to be useful, it must be both general and specific, but more importantly it must be extensible and adaptable. It is through the integration of new tools and actions that TASK will truly support collaborative work.

2.1.4 Keys

Clearly, unrestricted access to all tools by all actors may not be desired, and therefore I define the notion of a ***key*** to facilitate access restrictions within TASK. Keys are assigned on a per-action basis; that is, in order to perform a locked action on an object, an actor must hold the key with which it has been locked. In the event that an action is locked by more than one key, only one of these keys is needed to invoke the action. Actions can be locked or unlocked, and keys can be granted or revoked, dynamically according to specific permission or privilege needs. For example, the actions to destroy and to lock an action of an object could be locked, and the key granted to an actor when the object is created. This key could then, in a sense, represent a form of ownership within the framework, where the holder of the key is implicitly the object's "owner".

Interactions 2 and 4 from our electronic classroom case study illustrate the use of keys. It can be assumed that the instructor was the actor that created the file and projector objects, and hence represents their "owner". In interaction 2, the instructor gives all of the students access to the file by unlocking its read action, thus making it available for download. In interaction 4, the instructor gives one student exclusive access to the projector by locking its actions and granting the key to the student. In both cases, the instructor may also have to modify previously granted privileges by revoking keys from other actors. The use of keys in this manner provides a flexible, yet powerful, mechanism of limiting the behaviors of actors within TASK.

2.2 Constraints in TASK

In describing conceptual models, it is helpful to use class diagrams (described in more detail in Chapter 3) to show the relationships that exist among the types of objects in a system. In general, there are two kinds of relationships: associations and subtypes. *Associations* represent conceptual relationships between classes in which each class plays a distinct role and for which each role has its own multiplicity (i.e. how many objects participate in the given relationship). *Subtypes* represent generalization relationships in which instances of the subtype (or child class) are also, by definition, instances of the supertype (or parent class). Class diagrams also describe the attributes and operations for the various kinds of objects. *Attributes* represent properties possessed by the class, whereas *operations* represent the processes that a class knows how to carry out. [Fowler 1997]

Figure 1 presents a class diagram using the Unified Modeling Language (see Chapter 3), which shows the relationships, attributes, and operations for actors, scopes, tools, and keys. Note that three supertypes (`Object`, `Contextual Object`, and `Context`) have been introduced to factor out characteristics that are common to two or more types of object. Specifically, `Key` is a subtype of `Object`, `Tool` is a subtype of `Contextual Object`, and `Actor` and `Scope` are subtypes of `Context`, which is in turn a subtype of `Contextual Object`. Associations exist between `Contextual Object` and `Key` (via a `Lock` association type), between `Contextual Object` and `Context`, between `Actor` and `Scope`, and between `Actor` and `Key`. All objects have name and description attributes. Operations for `Actor` include `create`, `describe`, `destroy`, `focalize`, lock, and unlock. Operations for Scope include describe, destroy, enter, exit, lock, and unlock. Operations for Tool include describe, destroy, drop, lock, take, and unlock. Operations for Key include describe, destroy, grant, and revoke.

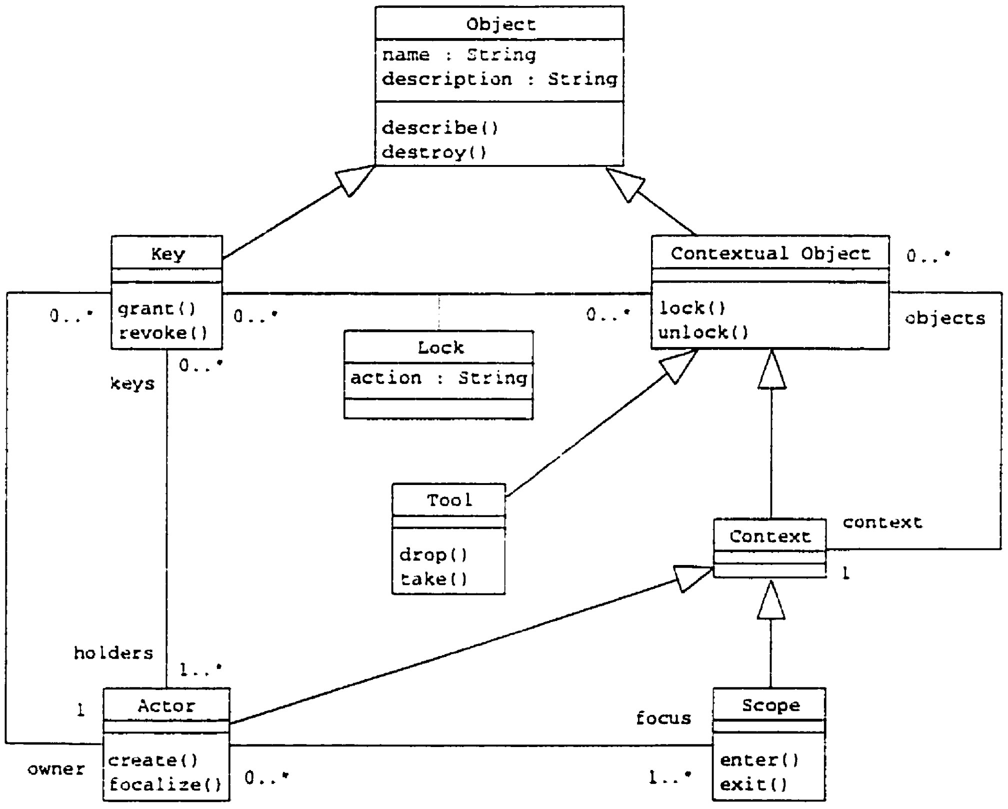


Figure 1 Objects in TASK.

Much of what a class diagram does is indicate constraints. A *constraint* is a restriction on one or more values of (part of) a model or system. The most common types of constraints are invariants, preconditions, and postconditions. An *invariant* represents a restriction on a class, type, or interface, and specifies conditions that must always hold true for all instances of that class, type, or interface. Both preconditions and postconditions represent restrictions on operations or methods. *Preconditions* specify conditions that must be true before an operation can be executed, whereas *postconditions* specify conditions that must be true immediately after the execution of an operation. [Warmer & Kleppe 1999a]

Although relationships and associations do much in the way of specifying constraints, they cannot possibly indicate every constraint. The Object Constraint Language (OCL) is an industry-standard textual language for describing constraints on object-oriented models. Using objects and object properties as its building blocks, the OCL defines basic types and operations that can be combined with user-defined model types to specify invariants, preconditions, and postconditions of a system. All OCL expressions are defined within a specific context: the context of an invariant is a class, interface, or type, whereas the context for preconditions and postconditions is an operation or a method. [Warmer & Kleppe 1999b]

In TASK, operations that can be directly invoked by actors are referred to as ***actions***. Actions are the unit of event atomicity in TASK. A series of actions requested of actors, scopes, tools, and keys by an actor represents an activity in which that actor is engaged. Actions will be defined for the various kinds of objects by applications that extend the TASK framework. The following sections describe the invariants of, and constraints on the predefined actions for, actors, scopes, tools, and keys in OCL using the following conventions:

- OCL keywords are in bold, although this is not part of the formal syntax
- The first line denotes the context of the constraint (type, class, interface, or operation).
- The second and subsequent lines contain the actual constraint(s) being described.

- Invariant, precondition, and postcondition expressions are preceded by identifiers **invariant:**, **pre:**, and **post:**, respectively.

2.2.1 Constraints on Actors

Since an actor is a kind of contextual object, its context (a scope) must include the actor as one of its objects; an actor is also a kind of context, so the context for all of its objects (tools) must be the actor; all of an actor's keys must include the actor as one if its holders; an actor's focus must include its context; all of the scopes in an actor's focus must include the actor as one of its objects. These invariants can be expressed using OCL as follows:

```
context Actor
invariant: context.objects->includes( self ) and objects->forAll( oclIsKindOf(
Tool ) and context = self ) and keys->forAll( holders->includes( self ) ) and
focus->includes( context ) and focus->forAll( objects->includes( self ) )
```

The `create` action on actors allows the requester to create a new actor, scope, tool, or key. In order for this action to be invoked, the actor must either be the requester itself or its focus must include the requester's context, the requester must hold at least one of the keys that locks the `create` action (if any), and the given name must be different from that of any existing actor, scope, tool, or key. As a result, a new object with the given name and description is added to TASK; if it is a key, its owner is the requester, otherwise (if it is an actor, scope, or tool) its context is the same as the requester's context. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::create( aRequester : Actor, aClassName : String, aName : String,
aDescription : String )
pre: ( self = aRequester or focus->includes( aRequester.context ) ) and ( lock-
>select( action = 'create' )->notEmpty implies lock->exists( action = 'create'
and key.holders->includes( aRequester ) ) ) and OclType.allInstances->exists(
name = aClassName ) and Tool.allInstances->forAll( name <> aName ) and
Actor.allInstances->forAll( name <> aName ) and Scope.allInstances->forAll( name
<> aName ) and Key.allInstances->forAll( name <> aName )
post: if OclType.allInstances->select( name = aClassName )->allSupertypes-
>includes( Key ) then ( OclType.allInstances->select( name = aClassName )-
>allInstances->exists( name = aName and description = aDescription and owner =
aRequester ) ) else ( OclType.allInstances->select( name = aClassName )-
>allInstances->exists( name = aName and description = aDescription and context =
aRequester.context ) ) endif
```

The `describe` action on actors allows the requester to change the description of an actor. In order for this action to be invoked, the actor must either be the requester itself or its focus must include the requester's context, the requester must hold at least one of the keys that locks the `describe` action (if any), and the new description must be different from the actor's existing description. As a result, the actor's description is changed to the new description. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::describe( aRequester : Actor, aDescription : String )
pre: ( self = aRequester or focus->includes( aRequester.context ) ) and ( lock-
>select( action = 'describe' )->notEmpty implies lock->exists( action =
'describe' and key.holders->includes( aRequester ) ) ) and description <>
aDescription
post: description = aDescription
```

The `destroy` action on actors allows the requester to destroy an actor. In order for this action to be invoked, the actor must not be the requester itself and its focus must include the requester's context, the requester must hold at least one of the keys that locks the `destroy` action (if any), the actor's objects must be empty, and the actor's keys must be empty. As a result, the actor (and any associations with it) is removed from TASK. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::destroy( aRequester : Actor )
pre: ( self <> aRequester and focus->includes( aRequester.context ) ) and (
lock->select( action = 'destroy' )->notEmpty implies lock->exists( action =
'destroy' and key.holders->includes( aRequester ) ) ) and objects->isEmpty and
keys->isEmpty
post: not Actor.allInstances->includes( self ) and Scopes.allInstances->forAll(
not objects.includes( self ) ) and Key.allInstances->forAll( not lock->exists(
object = self ) )
```

The `focalize` action on actors allows the requester to change the context for an actor. In order for this action to be invoked, the actor must either be the requester itself or its focus must include the requester's context, the requester must hold at least one of the keys that locks the `focalize` action (if any), the actor's focus must include the new context, and the new context must be different from the actor's existing context. As a result, the actor's context is changed to the new context. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::focalize( aRequester : Actor, aContext : Scope )
pre: ( self = aRequester or focus->includes( aRequester.context ) ) and ( lock-
>select( action = 'focalize' )->notEmpty implies lock->exists( action =
'focalize' and key.holders->includes( aRequester ) ) ) and focus->includes(
aContext ) and context <> aContext
post: context = aContext
```

The `lock` action on actors allows the requester to lock one of an actor's actions with a key. In order for this action to be invoked, the actor must either be the requester itself or its focus must include the requester's context, the requester must hold at least one of the keys that locks the `lock` action (if any) as well as the key being used to lock the specified action, and the action must not already be locked by the given key. As a result, a lock for the specified action on the actor is added with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::lock( aRequester : Actor, anAction : String, aKey : Key )
pre: ( self = aRequester or focus->includes( aRequester.context ) ) and ( lock-
>select( action = 'lock' )->notEmpty implies lock->exists( action = 'lock' and
key.holders->includes( aRequester ) ) ) and aRequester.keys->includes( aKey )
and not lock->exists( action = anAction and key = aKey )
post: lock->exists( action = anAction and key = aKey )
```

The `unlock` action on actors allows the requester to unlock one of an actor's actions with a key. In order for this action to be invoked, the actor must either be the requester itself or its focus must include the requester's context, the requester must hold at least one of the keys that locks the `unlock` action (if any) as well as the key being used to unlock the specified action, and the action must be locked by the given key. As a result, a lock for the specified action on the actor is removed with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Actor::unlock( aRequester : Actor, anAction : String, aKey : Key )
pre: ( self = aRequester or focus->includes( aRequester.context ) ) and ( lock-
>select( action = 'unlock' )->notEmpty implies lock->exists( action = 'unlock'
and key.holders->includes( aRequester ) ) ) and aRequester.keys->includes( aKey
) and lock->exists( action = anAction and key = aKey )
post: not lock->exists( action = anAction and key = aKey )
```

2.2.2 Constraints on Scopes

Since a scope is a kind of contextual object, its context (a scope) must include the scope as one of its objects; a scope is also a kind of context, so for all of its objects (actors, scopes, or tools), if the object is an actor its focus must include the scope, otherwise its context must be the scope. These invariants can be expressed using OCL as follows:

```
context Scope
invariant: context.objects->includes( self ) and objects->forAll( if
oclIsKindOf( Actor ) then focus.includes( self ) else context = self endif )
```

The `describe` action on scopes allows the requester to change the description of a scope. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `describe` action (if any), and the new description must be different from the scope's existing description. As a result, the scope's description is changed to the new description. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::describe( aRequester : Actor, aDescription : String )
pre: context = aRequester.context and ( lock->select( action = 'describe' )-
>notEmpty implies lock->exists( action = 'describe' and key.holders->includes(
aRequester ) ) ) and description <> aDescription
post: description = aDescription
```

The `destroy` action on scopes allows the requester to destroy a scope. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `destroy` action (if any), and the scope's objects must be empty. As a result, the scope (and any associations with it) is removed from TASK. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::destroy( aRequester : Actor )
pre: context = aRequester.context and ( lock->select( action = 'destroy' )-
>notEmpty implies lock->exists( action = 'destroy' and key.holders->includes(
aRequester ) ) ) and objects->isEmpty
post: not Scope.allInstances->includes( self ) and Scope.allInstances->forAll(
not objects->includes( self ) ) and Key.allInstances->forAll( not lock->exists(
object = self ) )
```

The `enter` action on scopes allows the requester to enter a scope. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `enter` action (if any), and the scope's objects must not already include the requester. As a result, the reqeuster is added as one of the scope's objects. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::enter( aRequester : Actor )
pre: context = aRequester.context and ( lock->select( action = 'enter' )-
>notEmpty implies lock->exists( action = 'enter' and key.holders->includes(
aRequester ) ) ) and not objects->includes( aRequester )
post: objects->includes( aRequester )
```

The `exit` action on scopes allows the requester to exit a scope. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `exit` action (if any), the scope's objects must not include any of the scopes in the requester's focus, and the scope's objects must include the requester. As a result, the requester is removed as one of the scope's objects. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::exit( aRequester : Actor )
pre: context = aRequester.context and ( lock->select( action = 'exit' )-
>notEmpty implies lock->exists( action = 'exit' and key.holders->includes(
aRequester ) ) ) and aRequester.focus->forAll( scope | not objects->includes(
scope ) ) and objects->includes( aRequester )
post: not objects->includes( aRequester )
```

The `lock` action on scopes allows the requester to lock one of a scope's actions with a key. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `lock` action (if any) as well as the key being used to lock the specified action, and the action must not already be locked by the given key. As a result, a lock for the specified action on the scope is added with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::lock( aRequester : Actor, anAction : String, aKey : Key )
pre: context = aRequester.context and ( lock->select( action = 'lock' )-
>notEmpty implies lock->exists( action = 'lock' and key.holders->includes(
aRequester ) ) ) and aRequester.keys->includes( aKey ) and not lock->exists(
action = anAction and key = aKey )
post: lock->exists( action = anAction and key = aKey )
```

The `unlock` action on scopes allows the requester to unlock one of a scope's actions with a key. In order for this action to be invoked, the scope's context must be the same as the requester's context, the requester must hold at least one of the keys that locks the `unlock` action (if any) as well as the key being used to unlock the specified action, and the action must be locked by the given key. As a result, a lock for the specified action on the scope is removed with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Scope::unlock( aRequester : Actor, anAction : String, aKey : Key )
pre: context = aRequester.context and ( lock->select( action = 'unlock' )-
>notEmpty implies lock->exists( action = 'unlock' and key.holders->includes(
aRequester ) ) ) and aRequester.keys->includes( aKey ) and lock->exists( action
= anAction and key = aKey )
post: not lock->exists( action = anAction and key = aKey )
```

2.2.3 Constraints on Tools

Since a tool is a kind of contextual object, its context (an actor or scope) must include the tool as one of its objects. This invariant can be expressed using OCL as follows:

```
context Tool
invariant: context.objects->includes( self )
```

The `describe` action on tools allows the requester to change the description of a tool. In order for this action to be invoked, the context of the tool must either be the same as the requester's context or the requester itself, the requester must hold at least one of the keys that locks the `describe` action (if any), and the new description must be different from the tool's existing description. As a result, the tool's description is changed to the new description. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::describe( aRequester : Actor, aDescription : String )
pre: ( context = aRequester or context = aRequester.context ) and ( lock-
>select( action = 'describe' )->notEmpty implies lock->exists( action =
'describe' and key.holders->includes( aRequester ) ) ) and description <>
aDescription
post: description = aDescription
```

The destroy action on tools allows the requester to destroy a tool. In order for this action to be invoked, the context of the tool must either be the same as the requester's context or the requester itself, and the requester must hold at least one of the keys that locks the destroy action (if any). As a result, the tool (and any associations with it) is removed from TASK. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::destroy( aRequester : Actor )
pre: ( context = aRequester or context = aRequester.context ) and ( lock-
>select( action = 'destroy' )->notEmpty implies lock->exists( action = 'destroy'
and key.holders->includes( aRequester ) ) )
post: not Tool.allInstances->includes( self ) and Actor.allInstances->forAll(
not objects->includes( self ) ) and Scope.allInstances->forAll( not objects-
>includes( self ) ) and Key.allInstances->forAll( not lock->exists( object =
self ) )
```

The drop action on tools allows the requester to put a tool down. In order for this action to be invoked, the context of the tool must be the requester itself, and the requester must hold at least one of the keys that locks the drop action (if any). As a result, the context for the tool is changed to be the same as the requester's context. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::drop( aRequester : Actor )
pre: context = aRequester and ( lock->select( action = 'drop' )->notEmpty
implies lock->exists( action = 'drop' and key.holders->includes( aRequester ) )
)
post: context = aRequester.context
```

The lock action on tools allows the requester to lock one of a tool's actions with a key. In order for this action to be invoked, the context of the tool must either be the same as the requester's context or the requester itself, the requester must hold at least one of the keys that locks the `lock` action (if any) as well as the key being used to lock the specified action, and the action must not already be locked by the given key. As a result, a lock for the specified action on the tool is added with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::lock( aRequester : Actor, anAction : String, aKey : Key )
pre: ( context = aRequester or context = aRequester.context ) and ( lock-
>select( action = 'lock' )->notEmpty implies lock->exists( action = 'lock' and
key.holders->includes( aRequester ) ) ) and aRequester.keys->includes( aKey )
and not lock->exists( action = anAction and key = aKey )
post: lock->exists( action = anAction and key = aKey )
```

The `take` action on tools allows the requester to pick a tool up. In order for this action to be invoked, the context of the tool must be the same as the requester's context, and the requester must hold at least one of the keys that locks the `take` action (if any). As a result, the context for the tool is changed to be the requester itself. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::take( aRequester : Actor )
pre: context = aRequester.context and ( lock->select( action = 'take' )-
>notEmpty implies lock->exists( action = 'take' and key.holders->includes(
aRequester ) ) )
post: context = aRequester
```

The `unlock` action on tools allows the requester to unlock one of a tool's actions with a key. In order for this action to be invoked, the context of the tool must either be the same as the requester's context or the requester itself, the requester must hold at least one of the keys that locks the `unlock` action (if any) as well as the key being used to unlock the specified action, and the action must be locked by the given key. As a result, a lock for the specified action on the tool is removed with the given key. These preconditions and postconditions can be expressed using OCL as follows:

```
context Tool::unlock( aRequester : Actor, anAction : String, aKey : Key )
pre: ( context = aRequester or context = aRequester.context ) and ( lock-
>select( action = 'unlock' )->notEmpty implies lock->exists( action = 'unlock'
and key.holders->includes( aRequester ) ) ) and aRequester.keys->includes( aKey
) and lock->exists( action = anAction and key = aKey )
post: not lock->exists( action = anAction and key = aKey )
```

2.2.4 Constraints on Keys

A key's holders must include its owner; all of a key's holders must include the key as one of its keys. These invariants can be expressed using OCL as follows:

```
context Key
invariant: holders->includes( owner ) and holders->forAll( keys->includes( self
) )
```

The `describe` action on keys allows the requester to change the description of a key. In order for this action to be invoked, the owner of the key must be the requester itself, and the new description must be different from the key's existing description. As a result, the key's description is changed to the new description. These preconditions and postconditions can be expressed using OCL as follows:

```
context Key::describe( aRequester : Actor, aDescription : String )
pre: owner = aRequester and description <> aDescription
post: description = aDescription
```

The `destroy` action on keys allows the requester to destroy a key. In order for this action to be invoked, the owner of the key must be the requester itself. As a result, the key (and any associations with it) is removed from TASK. These preconditions and postconditions can be expressed using OCL as follows:

```
context Key::destroy( aRequester : Actor )
pre: owner = aRequester
post: not Key.allInstances->includes( self ) and Tool.allInstances->forAll( not
lock->exists( key = self ) ) and Actor.allInstances->forAll( not lock->exists(
key = self ) and not keys->includes( self ) ) and Scope.allInstances->forAll(
not lock->exists( key = self ) )
```

The `grant` action on keys allows the requester to grant a key to an actor. In order for this action to be invoked, the owner of the key must be the requester itself, the actor's focus must include the requester's context, and the holders of the key must not already include the actor. As a result, the requester is added as one of the key's holders. These preconditions and postconditions can be expressed using OCL as follows:

```
context Key::grant( aRequester : Actor, aGrantee : Actor )
pre: owner = aRequester and aGrantee.focus->includes( aRequester.context ) and
not holders->includes( aGrantee )
post: holders->includes( aGrantee )
```

The `revoke` action on keys allows the requester to revoke a key from an actor. In order for this action to be invoked, the owner of the key must be the requester itself, the actor's focus must include the requester's context, and the holders of the key must include the actor. As a result, the requester is removed as one of the key's holders. These preconditions and postconditions can be expressed using OCL as follows:

```
context Key::revoke( aRequester : Actor, aRevokee : Actor )
pre: owner = aRequester and aRevokee.focus->includes( aRequester.context ) and
holders->includes( aRevokee )
post: not holders->includes( aRevokee )
```

3 DESIGN SPECIFICATION

The conceptual model of TASK, as introduced in Chapter 2, can be used to describe various scenarios of teaching in an electronic classroom. However, the framework is quite general; the case study presented here was meant to show its usefulness within a specific domain (education). Other applications of TASK range from simple ones such as file permissions on a Unix operating system, to rather complex ones such as collaborative work in a software development organization. This chapter presents the design specification for TASK by describing its object model using the UML and exploring the application of design patterns to facilitate object interactions within TASK.

3.1 TASK and the UML

The Unified Modeling Language (UML) is the industry standard language for visualizing, specifying, constructing, and documenting the artifacts of a software system. It fuses the concepts of the Booch, OMT (Object Modeling Technique), OOSE (Object-Oriented Software Engineering) methodologies, among others. Created by the primary authors of the original methods (Grady Booch, Jim Rumbaugh, and Ivar Jacobson, respectively), the UML focuses on a standard modeling language rather than a standard for tools and processes. It provides a common metamodel (a language for specifying a model) and notation which, together with guidelines for usage, integrate best industry practices to support any use-case driven, architecture centric, iterative and incremental approach. [Booch et al. 1999]

A UML diagram is a graphical projection of a collection of model elements, typically represented as a connected graph of arcs and vertices. Types of diagrams defined by the UML include class diagrams, object diagrams, use case diagrams, sequence diagrams, collaboration diagrams, statechart diagrams, activity diagrams, component diagrams, and deployment diagrams. [Booch et al. 1999]

Various aspects of the TASK framework's design will now be exposed using three of the graphical diagram types defined by the UML (use case diagrams, class diagrams, and sequence diagrams). Only elements of the UML semantics and notation that apply to these diagrams will be discussed here. For more information on the semantic and notational elements of the UML, the reader is referred to [Booch et al. 1999].

3.1.1 Use Case Diagrams

A UML use case diagram is a visual representation of the relationships between model elements such as use cases and actors. A *use case* is a sequence of actions performed by the system that yields an observable result to an actor. An *actor* in UML is an entity outside the system that interacts with use cases. Use case diagrams are used to specify or characterize the functionality and behavior of interactions between a system and external actors. [Booch et al. 1999]

A use case is represented as a hollow ellipse, below which the name of the use case is placed. An actor is represented by a stick figure, typically with the name of the actor located below the figure. Interactions between actors and use cases are represented as (unidirectional) associations between their respective representations. Use case diagrams for use cases involving the various kinds of objects in TASK will now be considered. [Booch et al. 1999]

Figure 2 presents a use case diagram for use cases involving objects. Use cases in this diagram include *Describe an object* and *Destroy an object*. These correspond to the describe and destroy actions defined on actors, scopes, tools, and keys, the semantics for which were described in Sections 2.2.1, 2.2.2, 2.2.3, and 2.2.4 of Chapter 2.

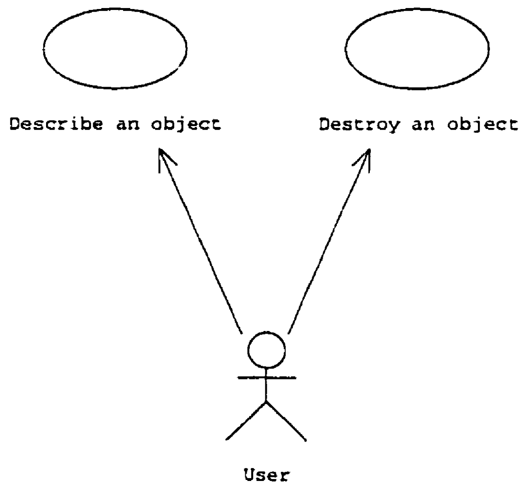


Figure 2 Use cases involving objects.

Figure 3 presents a use case diagram for use cases involving contextual objects. Use cases in this diagram include *Lock an action* and *Unlock an action*. These correspond to the lock and unlock actions defined on actors, scopes, and tools, the semantics for which were described in Sections 2.2.1, 2.2.2, and 2.2.3 of Chapter 2.

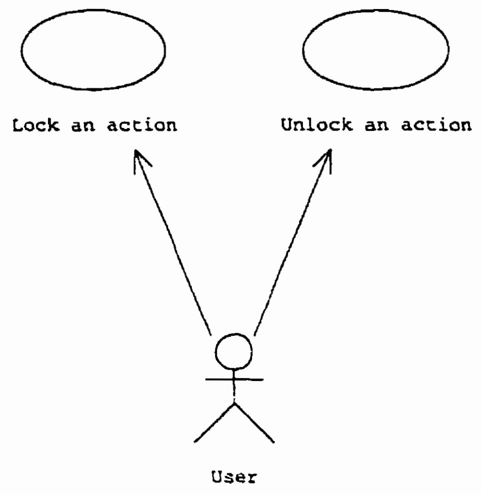


Figure 3 Use cases involving contextual objects.

Figure 4 presents a use case diagram for use cases involving actors. Use cases in this diagram include *Create an object* and *Focalize a scope*. These correspond to the `create` and `focalize` actions defined on actors, the semantics for which were described in Section 2.2.1 of Chapter 2.

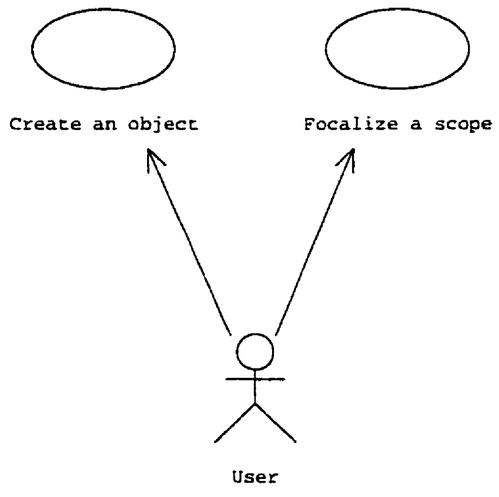


Figure 4 Use cases involving actors.

Figure 5 presents a use case diagram for use cases involving scopes. Use cases in this diagram include *Enter a scope* and *Exit a scope*. These correspond to the `enter` and `exit` actions defined on scopes, the semantics for which were described in Section 2.2.2 of Chapter 2.

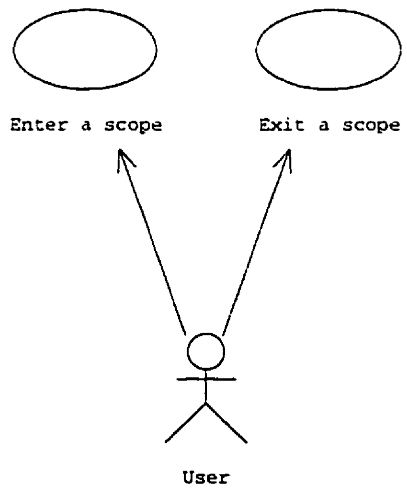


Figure 5 Use cases involving scopes.

Figure 6 presents a use case diagram for use cases in involving tools. Use cases in this diagram include *Drop a tool* and *Take a tool*. These correspond to the `drop` and `take` actions defined on tools, the semantics for which were described in Section 2.2.3 of Chapter 2.

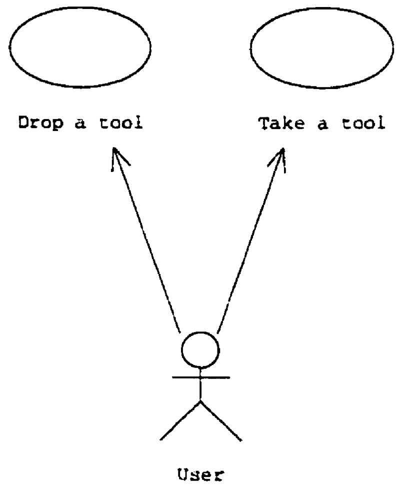


Figure 6 Use cases involving tools.

Figure 7 presents a use case diagram for use cases in involving keys. Use cases in this diagram include *Grant a key* and *Revoke a key*. These correspond to the `grant` and `revoke` actions defined on keys, the semantics for which were described in Section 2.2.4 of Chapter 2.

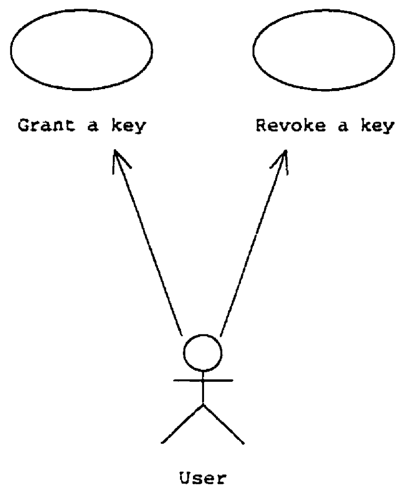


Figure 7 Use cases involving keys.

3.1.2 Class Diagrams

A UML class diagram is a visual representation of the relationships between model elements such as classes, interfaces, associations, aggregations, and generalizations. A *class* is the description of all objects with similar structure, behavior, and relationships. An *interface* is a collection of operations used to specify a service provided by a class or component. A *realization* is a semantic relationship between an interface and a class that realizes or implements it. An *association* in UML is a relationship that describes a set of semantic connections among tuples of objects. An *association class* is a modeling element that has both association and class properties; it can be seen as either a class with association properties or an association with class properties. An *aggregation* is form of association that represents a whole-part relationship between an aggregate and its component part(s). A *generalization* is an inheritance relationship between a more general element and a more specific element. Class diagrams are used to provide generic descriptions of systems. [Booch et al. 1999]

A class is represented as a solid rectangle with three compartments separated with horizontal lines. The top compartment contains the name of the class, following the syntax *Package-name::Class-name*, of which the package name is optional. The middle compartment lists the attributes of the class, following the syntax *visibility name : type-expression = initial-value { property-string }*, of which the visibility, type expression, initial value, and property string are optional. The bottom compartment lists the operations of the class, following the syntax *visibility name (parameter-list) : return-type-expression { property-string }*, of which the visibility, parameter list, return type expression, and property string are optional. An interface is represented as a small circle labeled with its name, or alternatively as a class with an <<*interface*>> stereotype in the name compartment. A realization is represented as a solid line connecting a class to an interface in its normal form (as a circle) or as a dotted arrow between a class and an interface in its expanded form (as a class). An association is represented as a solid line connecting two class symbols with an optional name. The end of an association where it connects to a class is called an *association role*, and may or may not have a name. An association role may also have a multiplicity, which follows the syntax *lower-bound .. upper-bound.* An association class is represented as a class symbol attached by a dashed line to an association. An aggregation is represented by attaching a hollow diamond where an association meets the class that represents the aggregate. A generalization is represented as a solid line from the more specific element to the more general element, with a hollow triangle where the line meets the more general element. Class diagrams for the classes and interfaces in TASK will now be considered. [Booch et al. 1999]

Figure 8 presents a class diagram depicting the interfaces in TASK and the classes that realize them. Interfaces include `TASKObject`, `TASKContextualObject`, `TASKContext`, Actor, Scope, Tool, and Key. The classes that realize these interfaces are TASKObjectImpl, TASKContextualObjectImpl, TASKContextImpl, ActorImpl, ScopeImpl, ToolImpl, and KeyImpl, respectively.

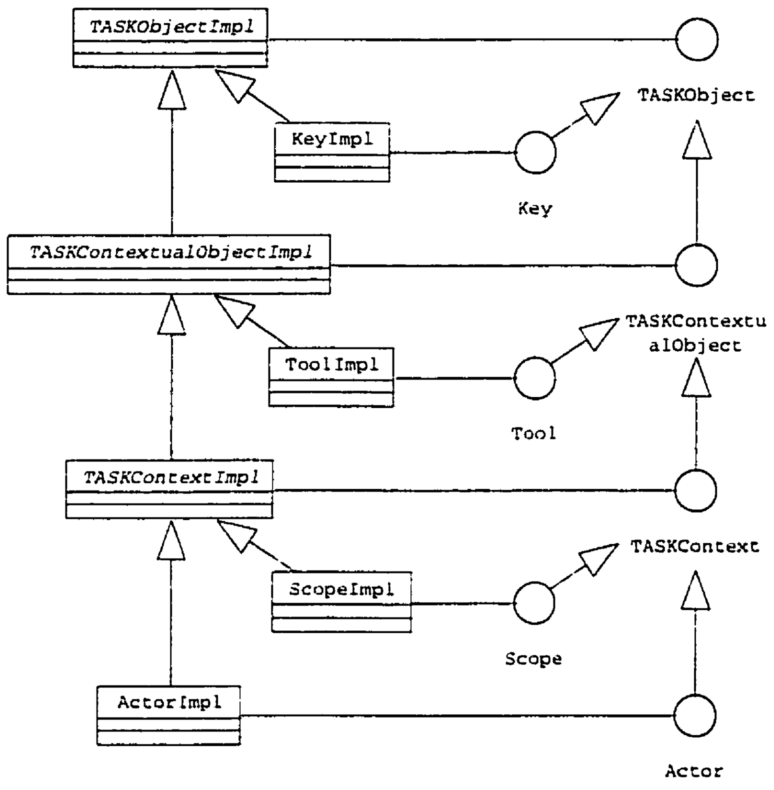


Figure 8 Classes and interfaces in TASK.

Figure 9 presents a class diagram depicting the interfaces and associated operations in TASK. These interfaces correspond directly to the objects described as part of the conceptual model presented in Chapter 2. Likewise, half of the operations provided by these interfaces correspond with the actions introduced in Chapter 2. The description() and name() operations are included to provide access to the values of description and name attributes of objects. The `isActor()`, `isScope()`, `isTool()`, and `isKey()` operations indicate whether an object is an actor, scope, tool, or key. The `actions()` operation returns the names of the actions defined on a given object. The `context()` and `objects()` operations are included to provide access to the roles of the association between contextual objects and contexts. The `focus()` operation provides access to the role played by scopes in their association with actors. Finally, the `keys()`, `holders()`, and `owner()` operations provide access to the roles in the associations between actors and keys.

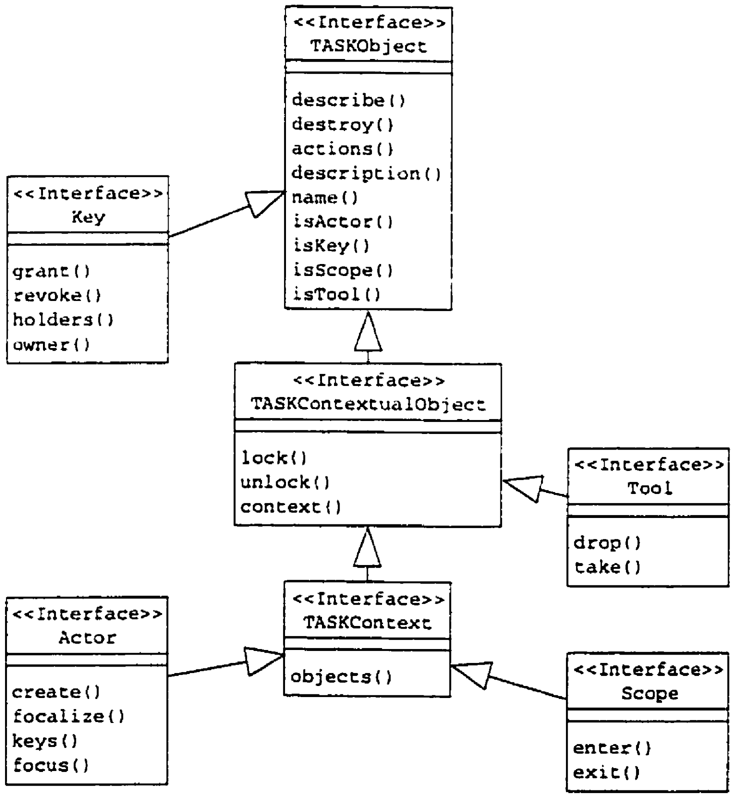


Figure 9 Interfaces in TASK.

Figure 10 presents a class diagram depicting the classes in TASK, along with their attributes and operations. Most of these classes, along with their associated operations, realize the interfaces that are described above. One additional class, the `TASKLock` class, with `action`, `key`, and `object` attributes, is introduced to model the association between contextual objects and keys.

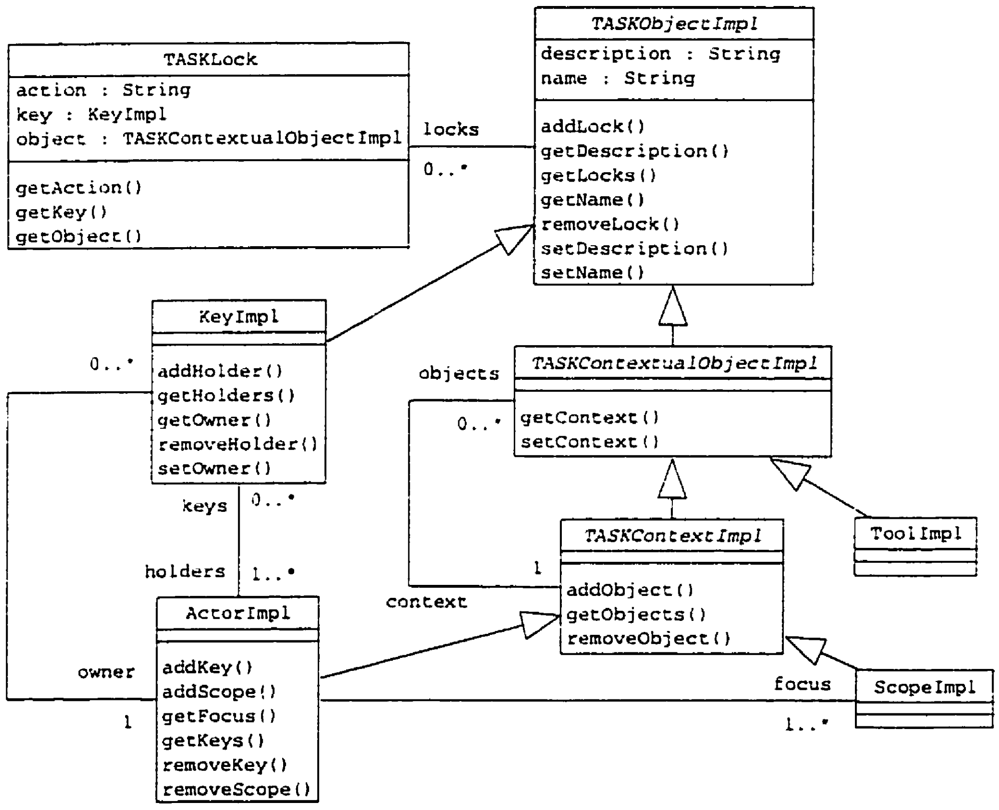


Figure 10 Classes in TASK.

3.1.3 Sequence Diagrams

A UML sequence diagram is a visual representation of the relationships between model elements such as objects, messages, and activations. An *object* is an instance of a class with a well-defined identity. A *message* is a communication between objects which results in some activity. An *activation* is the execution of a computational or algorithmic procedure. Sequence diagrams are used to trace the execution of an interaction in time. [Booch et al. 1999]

An object is represented as a dashed vertical line called the *lifeline*. A solid rectangle containing the name of the object, following the syntax *object-name : class-name* (of which the object name is optional) is placed at the top of the vertical line. A message is represented as a solid horizontal arrow between the lifelines of the sender and receiver objects. The arrow is labeled with the name of the message along with the values of its arguments, and may also include a sequence number indicating its order in the overall sequence. An activation is represented as a tall narrow rectangle on an object's lifeline, whose top is aligned with its initiation time and whose bottom is aligned with its completion time. Sequence diagrams for the pre-defined actions in TASK will now be considered. [Booch et al. 1999]

Figure 11 presents a sequence diagram for the `describe` action defined on `Actor`, `Scope`, `Tool`, and `Key`. Assuming the preconditions have been met, the object being described sends the `setDescription()` message to itself with the given description as an argument, thus changing the value of its description attribute to the specified description.

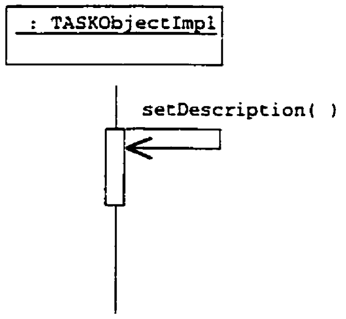


Figure 11 Message sequence for the `describe` action.

Figure 12 presents a sequence diagram for the `lock` action defined on `Actor`, `Scope`, and `Tool`. Assuming the preconditions have been met, the contextual object whose action is being locked creates a new lock with itself, the name of the given action, and the given key as attributes. It then sends the `addLock()` message to itself with the new lock as an argument, thus adding the lock to its set of locks. Finally, it sends the `addLock()` message to the given key with the new lock as an argument, thus adding the lock to the key's set of locks.

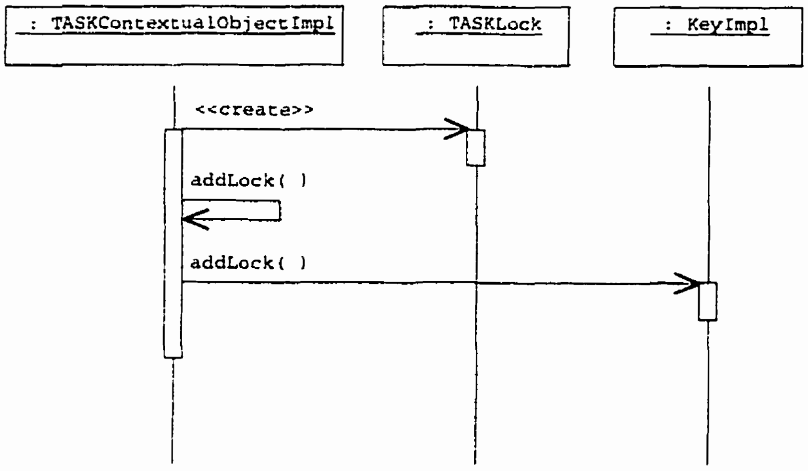


Figure 12 Message sequence for the `lock` action.

Figure 13 presents a sequence diagram for the `unlock` action defined on `Actor`, `Scope`, and `Tool`. Assuming the preconditions have been met, the contextual object whose action is being unlocked send the `removeLock()` message to the given key with the particular lock as an argument, thus removing the lock from the key's set of locks. It then sends the `removeLock()` message to itself with the particular lock as an argument, thus removing the lock from its set of locks.

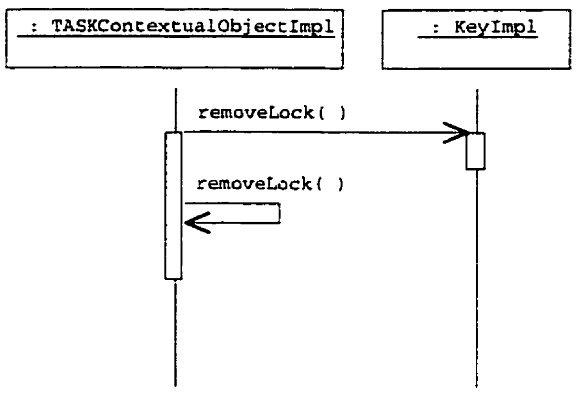


Figure 13 Message sequence for the `unlock` action.

Figure 14 presents a sequence diagram for the `create` action defined on `Actor`. Assuming the preconditions have been met, the actor creating the object creates a new object of the specified kind with the specified name and description as attributes. If the new object is a key, its owner is initialized to the creating actor itself, otherwise the context of the new object is initialized to the actor's context.

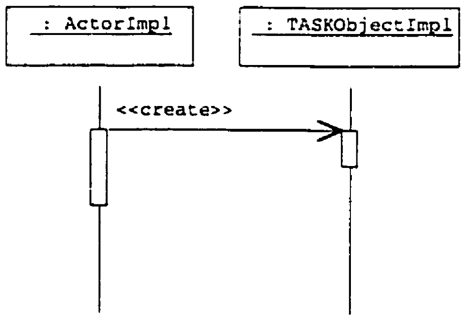


Figure 14 Message sequence for the `create` action.

Figure 15 presents a sequence diagram for the `destroy` action defined on `Actor`. Assuming the preconditions have been met, the actor being destroyed sends the `getFocus()` message to itself to retrieve the set of scopes representing its focus. It then sends the `removeObject()` message to each of these scopes with itself as an argument, thus removing itself from the scope's set of objects. Next, it sends the `getLocks()` message to itself to retrieve its set of locks. Finally, it sends the `getKey()` message to each of these locks to retrieve the key associated with the lock, and subsequently sends the `removeLock()` message to the key with the lock as an argument, thus removing the lock from the key's set of locks.

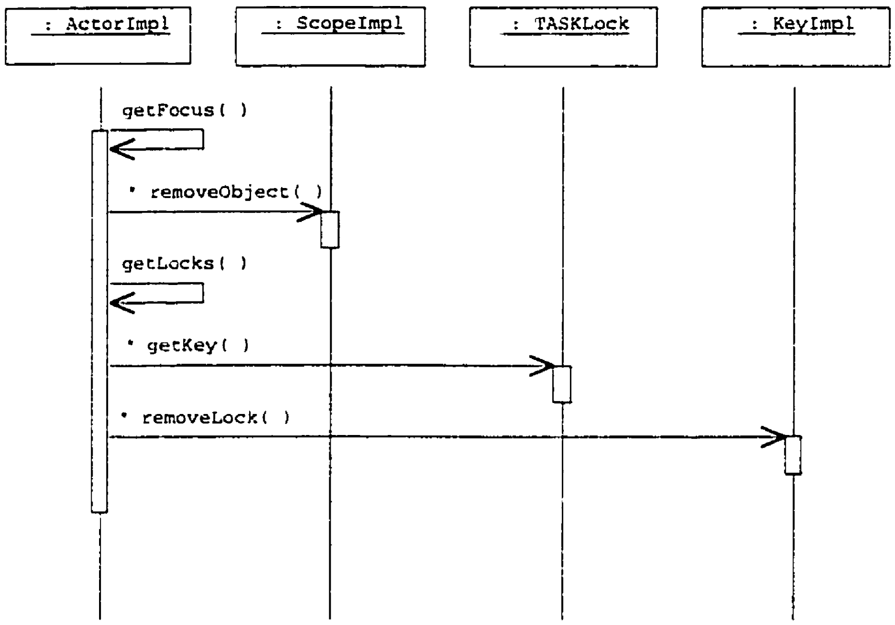


Figure 15 Message sequence for the `destroy` action (actors).

Figure 16 presents a sequence diagram for the `focalize` action defined on `Actor`. Assuming the preconditions have been met, the actor being focalized sends the `setContext()` message to itself with the given scope as an argument, thus changing its context to the specified scope.

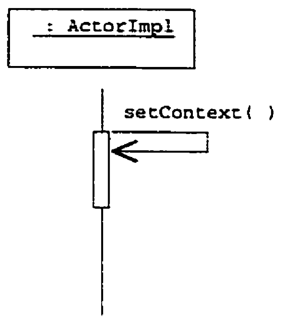


Figure 16 Message sequence for the `focalize` action.

Figure 17 presents a sequence diagram for the `destroy` action defined on `Scope`. Assuming the preconditions have been met, the scope being destroyed sends the `getContext()` message to itself to retrieve the scope representing its context. It then sends the `removeObject()` message to this scope with itself as an argument, thus removing itself from the scope's set of objects. Next, it sends the `getLocks()` message to itself to retrieve its set of locks. Finally, it sends the `getKey()` message to each of these locks to retrieve the key associated with the lock, and subsequently sends the `removeLock()` message to the key with the lock as an argument, thus removing the lock from the key's set of locks.

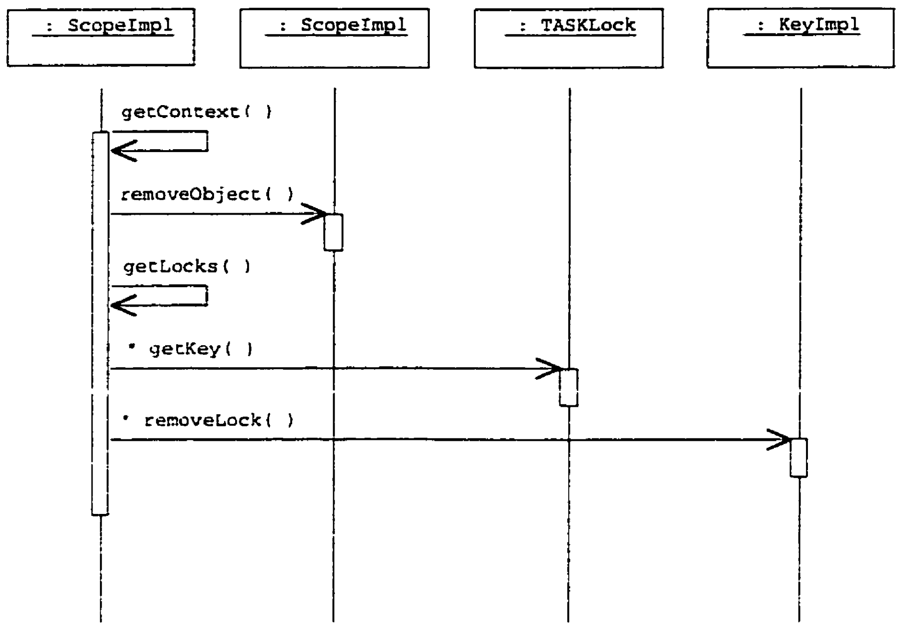


Figure 17 Message sequence for the destroy action (scopes).

Figure 18 presents a sequence diagram for the enter action defined on Scope. Assuming the preconditions have been met, the scope being entered sends the addObject() message to itself with the given actor as an argument, thus adding the actor to its set of objects. It then sends the addScope() message to the given actor, thus adding itself to the set of scopes representing the actor's focus.

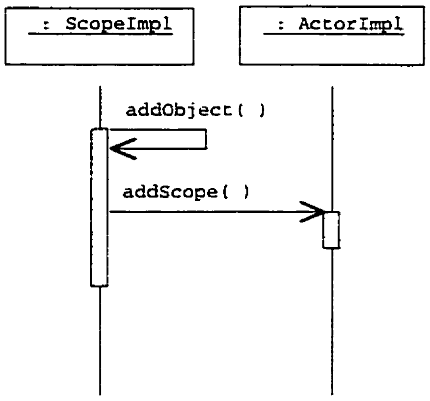


Figure 18 Message sequence for the `enter` action.

Figure 19 presents a sequence diagram for the `exit` action defined on `Scope`. Assuming the preconditions have been met, the scope being exited sends the `removeScope()` message to the given actor, thus removing itself from the set of scopes representing the actor's focus. It then sends the `removeObject()` message to itself with the given actor as an argument, thus removing the actor from its set of objects.

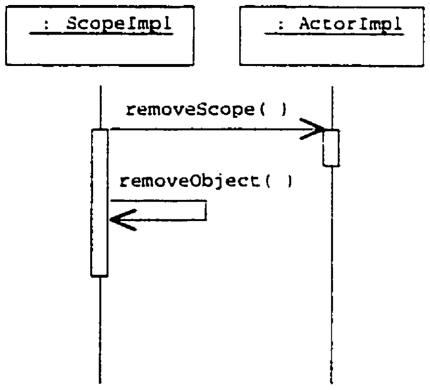


Figure 19 Message sequence for the `exit` action.

Figure 20 presents a sequence diagram for the destroy action defined on Tool. Assuming the preconditions have been met, the tool being destroyed sends the getContext() message to itself to retrieve the actor or scope representing its context. It then sends the removeObject() message to this actor or scope with itself as an argument, thus removing itself from the actor's or scope's set of objects. Next, it sends the getLocks() message to itself to retrieve its set of locks. Finally, it sends the getKey() message to each of these locks to retrieve the key associated with the lock, and subsequently sends the removeLock() message to the key with the lock as an argument, thus removing the lock from the key's set of locks.

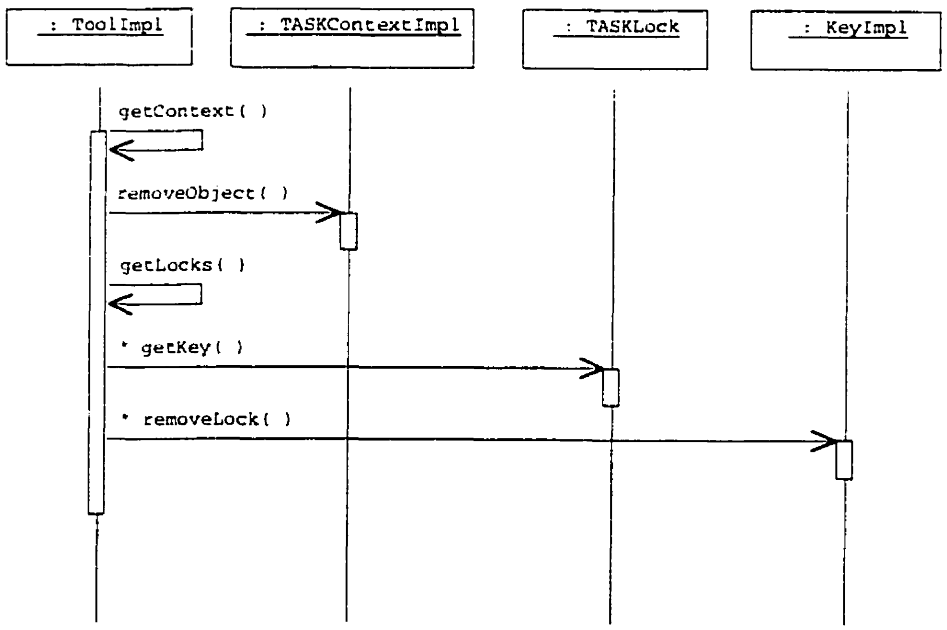


Figure 20 Message sequence for the destroy action (tools).

Figure 21 presents a sequence diagram for the drop action defined on Tool. Assuming the preconditions have been met, the tool being dropped sends the removeObject() message to the given actor with itself as an argument, thus removing itself from the actor's set of objects. It then sends the `setContext()` message to itself with the actor's context as an argument, thus changing its context to the actor's context. Finally, it sends the `addObject()` message to the scope representing its context with itself as an argument, thus adding itself to the scope's set of objects.

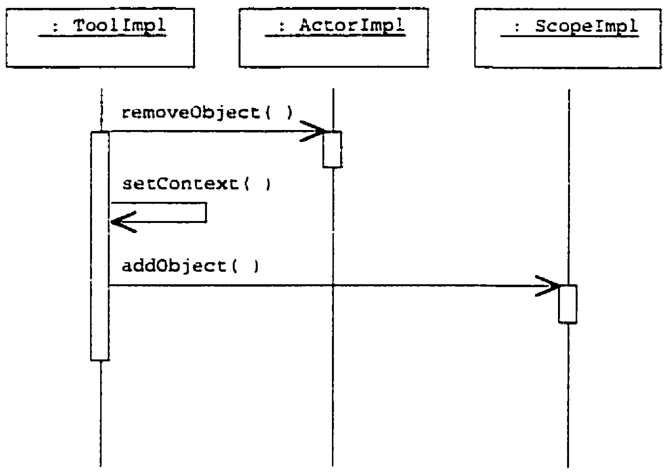


Figure 21 Message sequence for the `drop` action.

Figure 22 presents a sequence diagram for the `take` action defined on `Tool`. Assuming the preconditions have been met, the tool being taken sends the `removeObject()` message to the scope representing its context with itself as an argument, thus removing itself from the scope's set of objects. It then sends the `setContext()` message to itself with the given actor as an argument, thus changing its context to the actor. Finally, its sends the `addObject()` message to the given actor with itself as an argument, thus adding itself to the actor's set of objects.

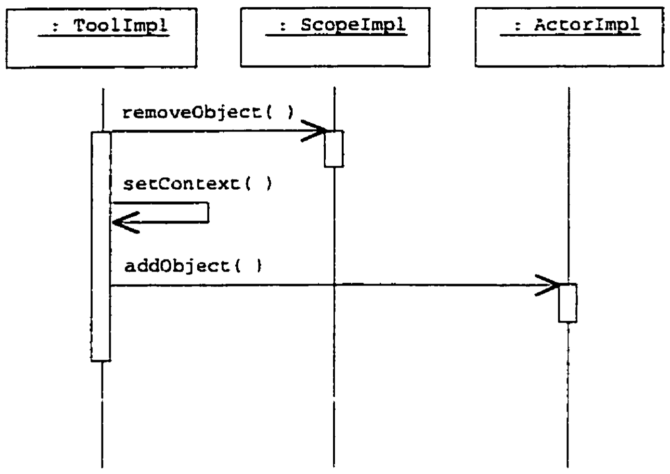


Figure 22 Message sequence for the take action.

Figure 23 presents a sequence diagram for the destroy action defined on Key. Assuming the preconditions have been met, the key being destroyed sends the getHolders() message to itself to retrieve the set of actors representing its holders. It then sends the removeKey() message to each of these actors with itself as an argument, thus removing itself from the actor's set of keys. Next, it sends the getLocks() message to itself to retrieve its set of locks. Finally, it sends the getObject() message to each of these locks to retrieve the contextual object associated with the lock, and subsequently sends the removeLock() message to the contextual object with the lock as an argument, thus removing the lock from the contextual object's set of locks.

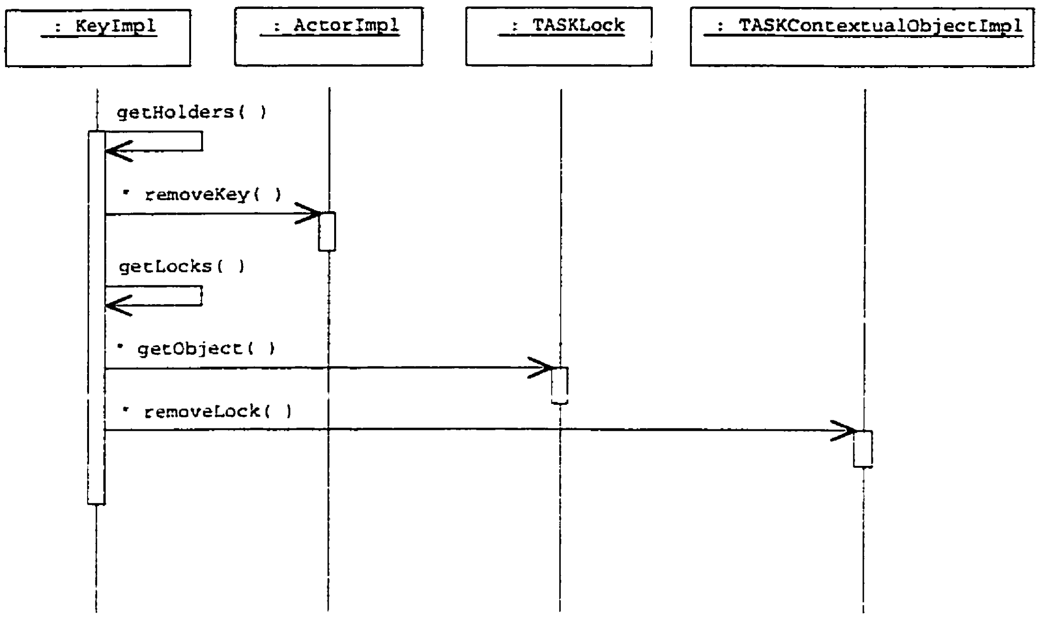


Figure 23 Message sequence for the destroy action (keys).

Figure 24 presents a sequence diagram for the grant action defined on Key. Assuming the preconditions have been met, the key being granted sends the addHolder() message to itself with the given actor as an argument, thus adding the actor to the set of actors representing its holders. It then sends the addKey() message to the given actor with itself as an argument, thus adding itself to the actor's set of keys.

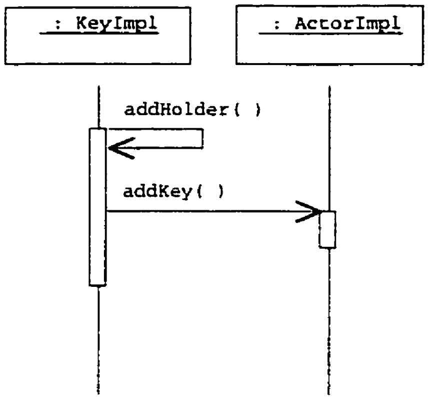


Figure 24 Message sequence for the `grant` action.

Figure 25 presents a sequence diagram for the `revoke` action defined on `Key`. Assuming the preconditions have been met, the key being revoked sends the `removeKey()` message to the given actor, thus removing itself from the actor's set of keys. It then sends the `removeHolder()` message to itself with the given actor as an argument, thus removing the actor from the set of actors representing its holders.

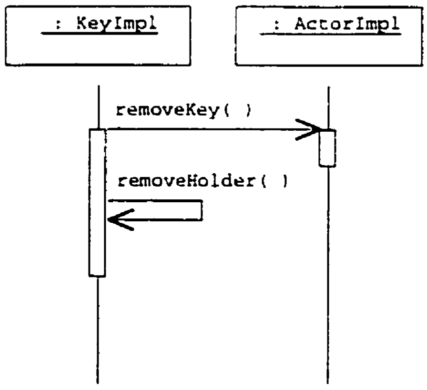


Figure 25 Message sequence for the `revoke` action.

3.2 Design Patterns in TASK

Design patterns are descriptions of communicating objects and classes customized to solve general design problems in specific contexts. A pattern systematically names, inspires, and explains a design that addresses recurring issues in object-oriented systems. Typically, it includes a description of the problem, the solution, when to apply the pattern, and the consequences of doing so. The inclusion of hints and examples of its application aids in the customization and implementation of the solution to solve the problem in a particular context. [Gamma et al. 1995]

Patterns can be classified in two ways: by purpose and by scope. *Purpose* reflects what a pattern does, and can be one of creational, structural, or behavioral. *Creational* patterns abstract the instantiation process, making a system independent of how its objects are created, composed, and represented. *Structural* patterns deal with the ways in which classes and objects are combined to form larger structures, and are especially useful in helping independently developed class libraries work together. *Behavioral* patterns deal with communication between objects and classes and focus on algorithms and the assignment of responsibilities to objects. The *scope* of a pattern refers to whether it applies to classes or objects. *Class* patterns are concerned with static relationships (association and subtype) between classes whereas *object* patterns involve dynamic relationships, which can be changed at run-time. [Gamma et al. 1995]

The use of patterns in the design of the communication mechanism in TASK is described in detail below. The goal here is to communicate the execution of actions on a given object to other objects in the environment. In general, when an action is invoked on a contextual object, each of the objects in the object's context need to be notified of the event's occurrence. To help accomplish this, I introduce an event class, TASK`Event`, which serves as the abstract parent for all events in TASK. Concrete subclasses of `TASKEvent` correspond to the actions in TASK, and include `ObjectDescribeEvent`, `ObjectDestroyEvent`, `ContextualObjectLockEvent`, `ContextualObjectUnlockEvent`, `ActorCreateEvent`, `ActorFocalizeEvent`, `ScopeEnterEvent`, `ScopeExitEvent`, `ToolDropEvent`, `ToolTakeEvent`, `KeyGrantEvent`, and `KeyRevokeEvent`. The suitability of three design patterns that could be used to model the desired communication mechanism will now be considered.

3.2.1 Mediator

The Mediator design pattern is an object behavioral pattern that defines an object to encapsulate how a set of objects interact, promoting loose coupling by preventing objects from referring to each other explicitly and allowing their interaction to vary independently. Participants in the pattern include the `Mediator`, which defines an abstract interface for communicating with `Colleague` objects, a `ConcreteMediator`, which implements cooperative behavior by coordinating its `Colleague` objects, and `Colleague` classes, each of which knows its `Mediator` and communicates with it whenever it would have instead communicated with another `Colleague` object. Figure 26 presents the generic structure of the Mediator design pattern. [Gamma et al. 1995]

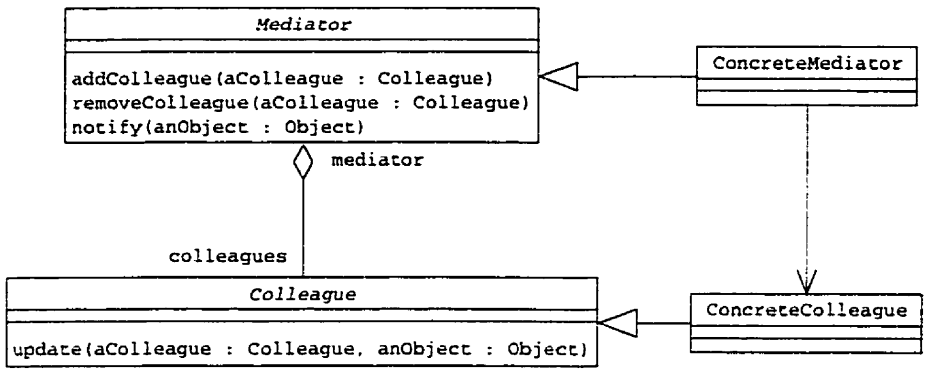
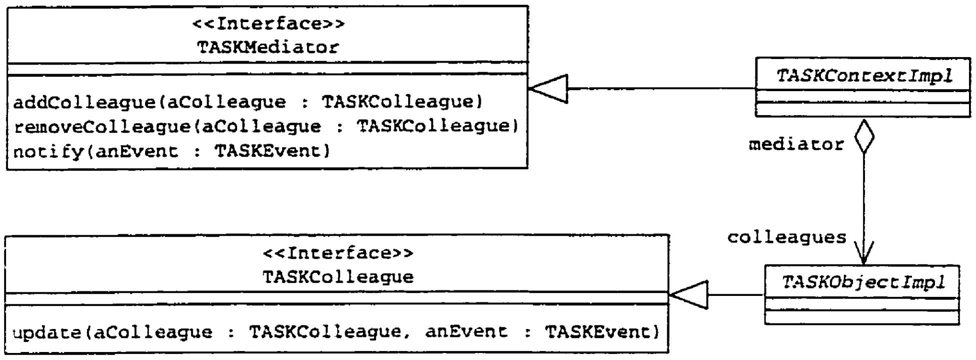

Mediator
addColleague(aColleague : Colleague)
removeColleague(aColleague : Colleague)
notify(anObject : Object)
mediator
colleagues
Colleague
update(aColleague : Colleague, anObject : Object)
ConcreteMediator
ConcreteColleague

Figure 26 The Mediator design pattern.

Figure 27 shows how the Mediator pattern could be applied in TASK. The `TASKMediator` and `TASKColleague` interfaces play the roles of the `Mediator` and `Colleague` participants, respectively. Through realization of these interfaces, the `TASKContextImpl` and `TASKObjectImpl` classes correspond to `ConcreteMediator` and `ConcreteColleague`. Using this mechanism, objects would be added and removed as colleagues of contexts as follows: actors would be added or removed as colleagues of scopes whenever scopes were entered or exited; scopes would be added or removed as colleagues of scopes whenever scopes were created or destroyed; tools would be added or removed as colleagues of actors and scopes whenever tools were dropped or taken; keys would be added or removed as colleagues of actors whenever keys were granted or revoked. A contextual object's mediator would be its context whereas a key's mediator would be its owner. Any time an action on an object is invoked, an event is created, and the object notifies its mediator, which in turn updates all of its colleagues with the event. One problem with this strategy, however, is that all colleagues of a particular mediator are notified of events regardless of whether they are interested in them.

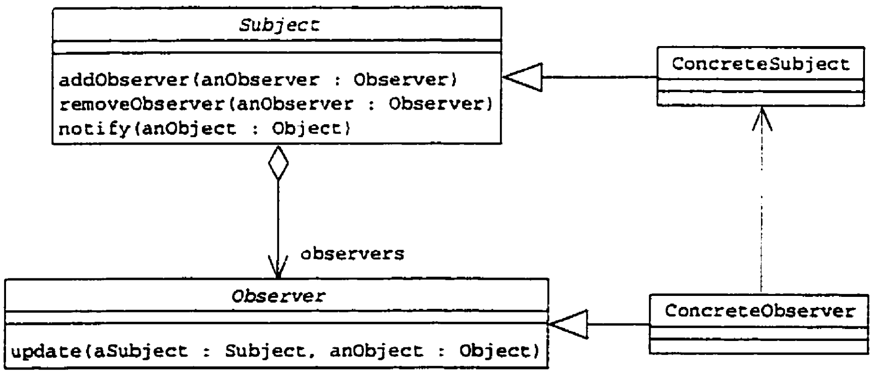


<<Interface>>
TASKMediator
addColleague(aColleague : TASKColleague)
removeColleague(aColleague : TASKColleague)
notify(anEvent : TASKEvent)
TASKContextImpl
mediator
colleagues
<<Interface>>
TASKColleague
update(aColleague : TASKColleague, anEvent : TASKEvent)
TASKObjectImpl

Figure 27 Application of Mediator in TASK.

3.2.2 Observer

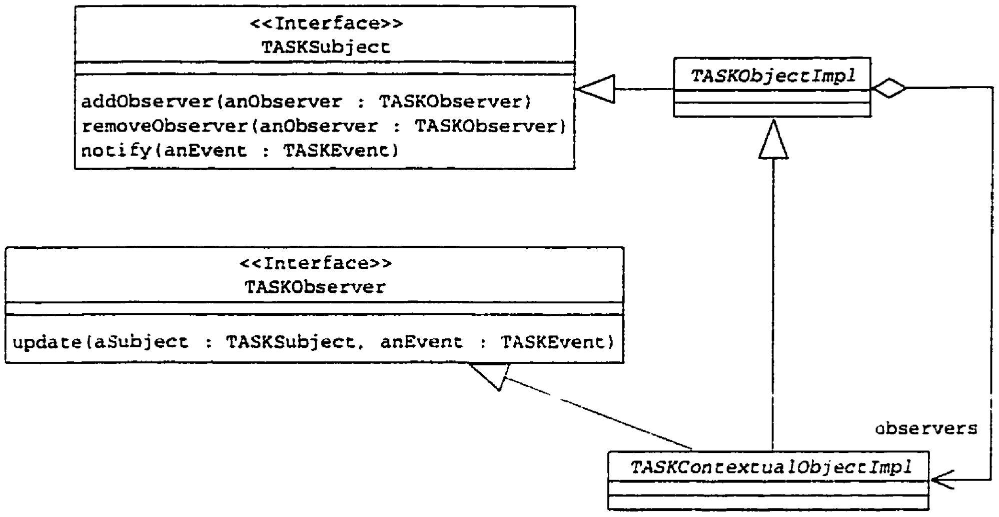
The Observer design pattern is an object behavioral pattern that defines a one-to-many dependency between objects, ensuring that when an object changes its state, all of its dependents are notified and updated. Participants in the pattern include the `Subject`, which knows its observers and provides an interface for attaching and detaching `Observer` objects, and an `Observer`, which defines an interface for updating it based on changes in the `Subject`. Figure 28 presents the generic structure of the Observer design pattern. [Gamma et al. 1995]



Subject
addObserver(anObserver : Observer)
removeObserver(anObserver : Observer)
notify(anObject : Object)
ConcreteSubject
observers
Observer
update(aSubject : Subject, anObject : Object)
ConcreteObserver

Figure 28 The Observer design pattern.

Figure 29 shows how the Observer pattern could be applied in TASK. The `TASKSubject` and `TASKObserver` interfaces play the roles of the `Subject` and `Observer` participants, respectively. Through realization of these interfaces, the `TASKObjectImpl` and `TASKContextualObjectImpl` classes correspond to `ConcreteSubject` and `ConcreteObserver`. Using this mechanism, contextual objects would be added and removed as observers of objects as follows: actors, scopes, and tools would be added or removed as observers of actors whenever scopes were entered or exited; actors, scopes, and tools would be added or removed as observers of scopes whenever scopes were created or destroyed; actors and scopes would be added or removed as observers of tools whenever tools were dropped or taken; actors would be added or removed as observers of keys whenever keys were granted or revoked. Any time an action is invoked on an object, an event is created, and the object updates each of its observers with the event. One problem with this strategy, however, is the burden placed on subjects of maintaining a list of observers and hence the coupling between subjects and observers.



<<Interface>>
TASKSubject
addObserver(anObserver : TASKObserver)
removeObserver(anObserver : TASKObserver)
notify(anEvent : TASKEvent)
TASKObjectImpl
<<Interface>>
TASKObserver
update(aSubject : TASKSubject, anEvent : TASKEvent)
observers
TASKContextualObjectImpl

Figure 29 Application of Observer in TASK.

3.2.3 Event Notifier

The Event Notifier pattern is an object behavioral pattern that enables components to react to the occurrence of events in other components without knowledge of one another, while also allowing dynamic component participation and introduction of new types of events. Participants in the pattern include `Event`, which represents the ancestor for all event types, `ConcreteEvent` objects, which represent specific events, `Publisher` objects, which publish the occurrence of events, `Subscriber`, which defines an abstract interface for all objects that are interested in events, `ConcreteSubscriber` objects, which register interest in particular events, `Filter`, which is responsible for weeding out events that are not of interest to a subscriber, and an `EventService`, which acts as an event broker between publishers and subscribers. Figure 30 presents the generic structure of the Event Notifier design pattern. [Gupta et al. 1998]

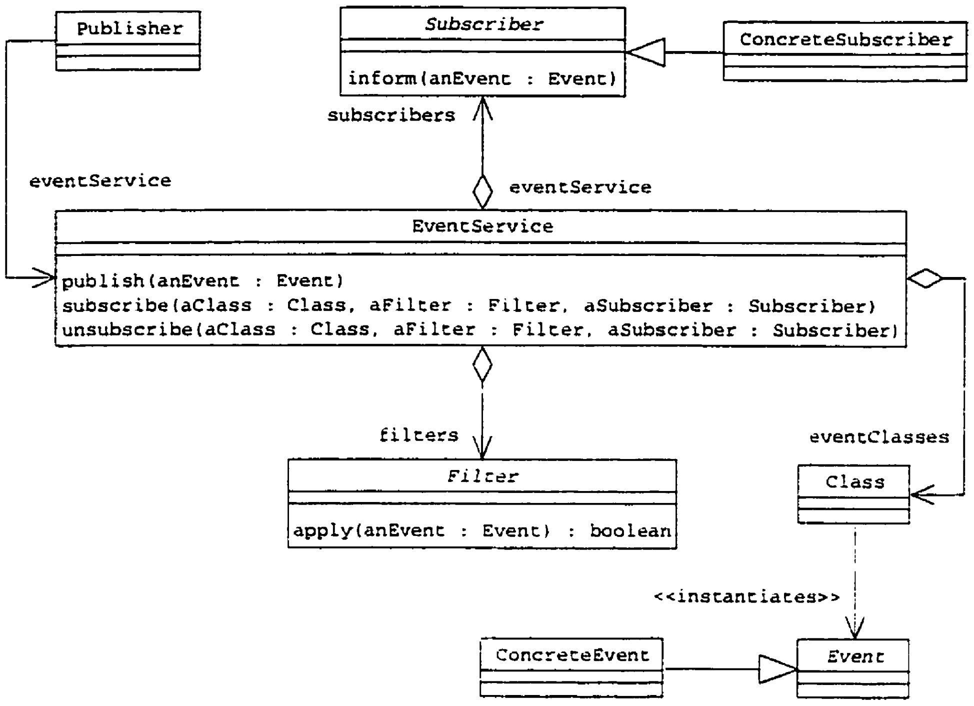
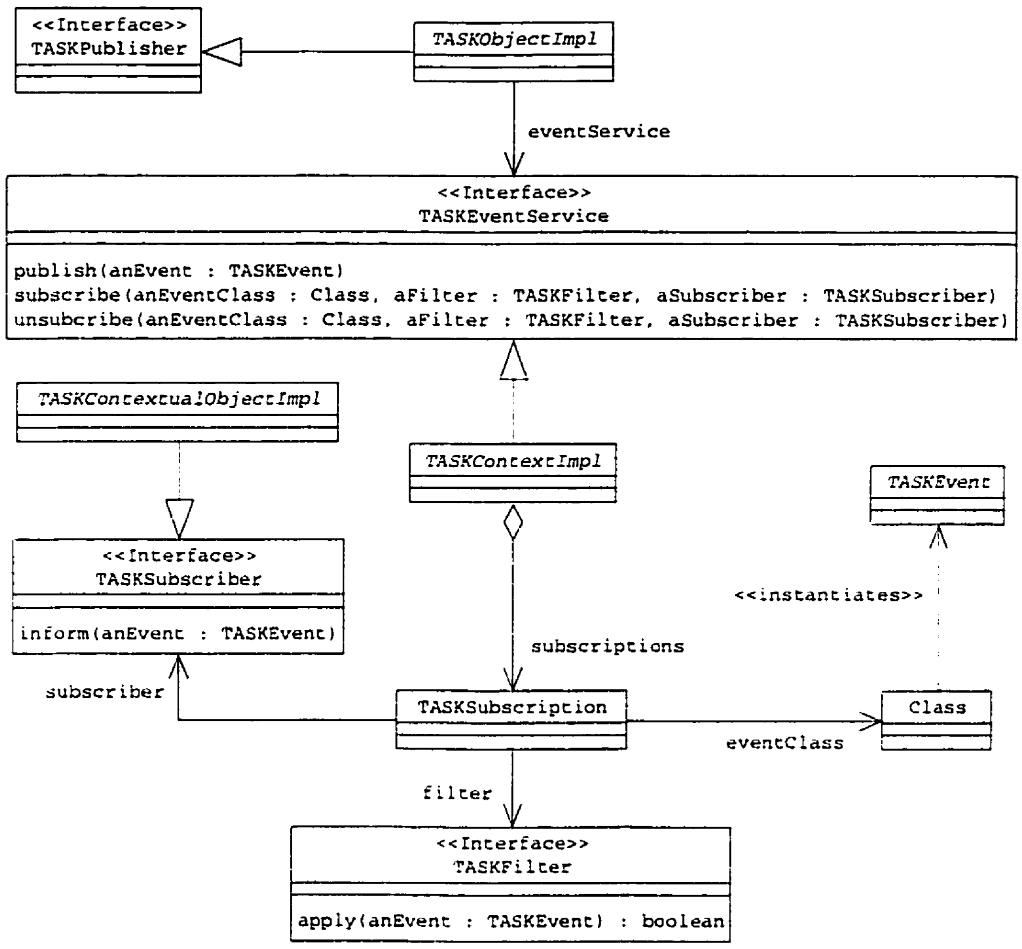


Figure 30 The Event Notifier design pattern.

Figure 31 shows how the Event Notifier pattern could be (and actually is) applied in TASK. Interfaces `TASKPublisher`, `TASKSubscriber`, `TASKFilter`, and `TASKEventService` play the roles of the `Publisher`, `Subscriber`, `Filter`, and `EventService` participants, respectively. Through realization of the `TASKSubscriber` interface, the `TASKContextualObjectImpl` class corresponds to `ConcreteSubscriber` and `TASKEvent`, along with all of its subclasses, corresponds to `Event` and `ConcreteEvent`. The `TASKSubcription` class is introduced to model the associations between `EventService` and `Subscriber`, `Class`, and `Filter`. Using this mechanism, contextual objects subscribe and unsubscribe to events dispatched by contexts as follows: actors subscribe or unsubscribe to events dispatched by scopes whenever scopes are entered or exited; scopes subscribe or unsubscribe to events dispatched by scopes whenever scopes are created or destroyed; and tools subscribe and unsubscribe to events dispatched by actors or scopes whenever tools are dropped or taken. In general, a contextual object's event service would be its context, an actor's event services would be the scopes in its focus, and a key's event services would be its holders. Any time an action on an object is invoked, an event is created, and the object publishes the event with its event service(s), which in turn informs all of its subscribers of the event. Event Notifier uses a best of both worlds approach that overcomes the problems of the previous two strategies – it facilitates subscription based on event type rather than publisher and allows publishers and subscribers to be varied independently of each other.



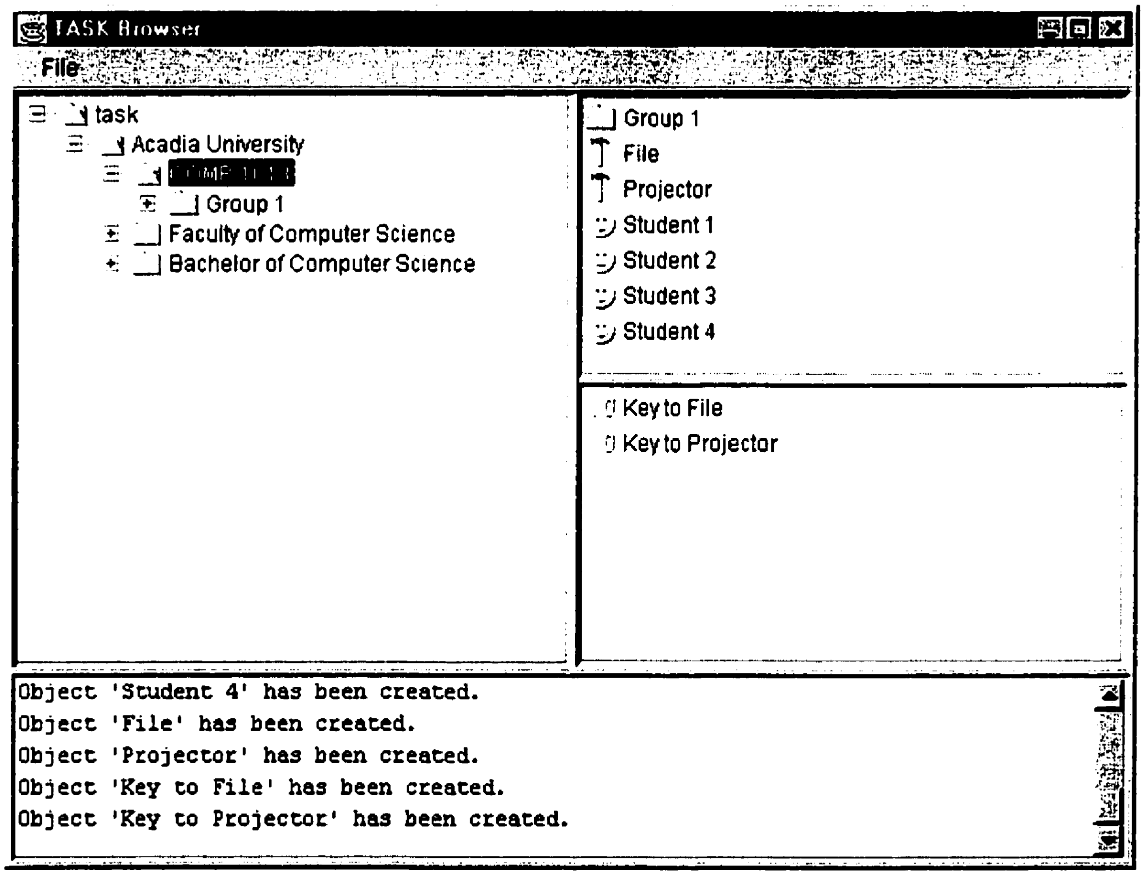
<<Interface>>
TASKPublisher
TASKObjectImpl
eventService
<<Interface>>
TASKEventService
publish(anEvent : TASKEvent)
subscribe(anEventClass : Class, aFilter : TASKFilter, aSubscriber : TASKSubscriber)
unsubcribe(anEventClass : Class, aFilter : TASKFilter, aSubscriber : TASKSubscriber)
TASKContextualObjectImpl
TASKContextImpl
TASKEvent
<<Interface>>
TASKSubscriber
inform(anEvent : TASKEvent)
<<instantiates>>
subscriptions
subscriber
TASKSubscription
Class
eventClass
filter
<<Interface>>
TASKFilter
apply(anEvent : TASKEvent) : boolean

Figure 31 Application of Event Notifier in TASK.

4 IMPLEMENTATION

In early days, known as the mainframe era, many users interacted with a single machine. Less expensive computing brought about the Personal Computer (PC) revolution, with computers on every desk, connected by a Local Area Network (LAN). Now, with the convergence of increasingly inexpensive computing and widespread connectivity through the Internet, we have entered the world of network computing, where every user will have access to many CPUs. With this shift in computing paradigm from host-centric to desktop-centric to network-centric came the evolution of the workspace. In the beginning there was little or no notion of workspace. PCs introduced the concept of a stand-alone, user-specific workspace. The advent of the network-centric age brought the shared workspace. From its inception, the Java language (see [Morrison 1997]) has embraced this network-centric view of the world, and as such, is an ideal platform for shared workspaces.

The Java programming language was used to implement TASK as per the design specified in Chapter 3. In addition to the framework itself, a simple GUI client, presented in Figure 32, was developed as a "proof of concept" for TASK. Modeled after the notion of a file browser, the TASK browser consists of a scopes pane (top left), which contains the hierarchy of scopes in TASK; a context pane (top right), which contains the objects in the user's current context; an actor pane (middle right), which contains the keys and tools currently held by the user; and a console pane (bottom), which contains a transcript of events. The TASK browser allows the user to interact with objects in TASK by invoking actions from pop-up menus associated with selected objects. For example, to grant the `Key to Projector` key to the `Student 1` actor, the user (connected to TASK as the `Instructor` actor) would select the key in the actor pane, click the right mouse button, and select the **`grant`** option. A dialog would appear, into which the user would enter the name of the grantee (**`Student 1`**). Upon accepting the entered value by clicking the **OK** button, the key would be granted to the `Student 1` actor and a description of the event would be appended to text in the console pane.



```
Object 'Student 4' has been created.
Object 'File' has been created.
Object 'Projector' has been created.
Object 'Key to File' has been created.
Object 'Key to Projector' has been created.
```

Figure 32 The TASK browser.

This chapter presents various aspects of the implementation of TASK in Java that proved both interesting and challenging to the author. More specifically, the JDK Collections API, implementation of constraints, object serialization, and remote objects will be considered in detail.

4.1 JDK Collections API

The Java Development Kit (JDK) Collections Application Programming Interface (API) is a new set of collection classes introduced as part of JDK 1.2 to be used as the basis for data structures in Java. A *collection* is a single object representing a group of other objects, referred to as elements of the collection. The new classes and interfaces, the root of which is the `Collection` interface, extend the facilities provided by the previously available utility classes such as `Vector` and `Hashtable`. The API can perhaps be partitioned as follows: collection interfaces, abstract implementations, concrete implementations, the `Collections` class, iteration, and array sorting and searching. [Hunt 1999]

Four interfaces, namely `Collection`, `Set`, `List`, and `Map`, comprise the core of the Collections API. The `Collection` interface defines the methods that all collections (except for `Map` collections) must implement and as such acts as the ancestor for virtually all collection objects. The `Set` interface is essentially the same as `Collection`, except that duplicates are not allowed in the set. The `List` interface represents a collection of elements in a specific sequence, whose order is defined by the order in which they are added to the list. The `Map` interface represents a set of associations, the elements of which may be unordered but must have a definite name or key.

The Collections API includes a set of abstract classes that provide basic implementations of many the methods defined in the interfaces they realize. The AbstractCollection class provides a skeletal implementation of the Collection interface, representing a collection of unordered objects, commonly referred to as a bag or multiset. The AbstractSet class, a direct subclass of AbstractCollection, realizes the Set interface and provides implementations for the equals() and hashCode() methods. The AbstractList class is also a direct subclass of AbstractCollection, and uses an array that is optimized for sequential access to maintain its internal data. The AbstractSequentialList class is basically the same as AbstractList except that its internal data structure is optimized for sequential rather than random access. The AbstractMap class, subclasses of which must implement the entries() method, provides an abstract implementation of its corresponding interface, Map.

The HashSet, ArraySet, ArrayList, LinkedList, HashMap, ArrayMap, and TreeMap classes provide general-purpose concrete implementations of the core interfaces in the Collections API. Each one inherits from an abstract implementation class, providing an example of the proper way to create concrete subclasses of the various collection types. Unlike their predecessors, Hashtable and Vector, these classes are unsynchronized, which results in greater performance (discussed in more detail later).

The Collections class was included in the Collections API to provide a range of static factory methods enabling data structures to be efficiently and effectively converted into collections. It also includes useful methods to sort and search collections, to find the minimum and maximum value in a collection, and to create immutable versions of modifiable collections.

The Enumeration interface is superceded by the more powerful Iterator interface in JDK 1.2. Together with the ListIterator interface, Iterator makes it possible to iterate over the contents of any of the collection classes, possibly modifying the collection while iteration is in progress. The hasMoreElements() method of Enumeration has been replaced by hasNext() defined on Iterator, just as the nextElement() method has been replaced by next().

Collections are used extensively throughout the implementation of TASK. Objects have a collection of locks, contexts (actors and scopes) have a collection of contextual objects (actors, tools, and scopes), actors have a collection of keys and a collection of scopes (representing their focus), and keys have a collection of actors (representing their holders). In particular, instances of the HashSet class are used to maintain the objects in these collections. For example, an instance of HashSet representing an actor's focus is created in a constructor for the ActorImpl class as follows:

```
...
focus = new HashSet();
...
```

The add() and addAll() methods are optional operations that are implemented by the HashSet class to add the specified element or elements to the set. The add() method is used in TASK any time a lock is added to an object's set of locks, a contextual object is added to a context's set of objects, a key is added to an actor's set of keys, a scope is added to an actor's focus, or an actor is added to a key's holders. For example, a scope is added to the instance of HashSet representing an actor's focus in the addScope() method of the ActorImpl class as follows:

```
public void addScope( ScopeImpl aScope )
{
    focus.add( aScope );
    return;
}
```

The remove() and removeAll() methods are optional operations that are implemented by the HashSet class to remove the specified element or elements from the set. The remove() method is used in TASK any time a lock is removed from an object's set of locks, a contextual object is removed from a context's set of objects, a key is removed from an actor's set of keys, a scope is removed from an actor's focus, or an actor is removed from a key's holders. For example, a scope is removed from the instance of HashSet representing an actor's focus in the removeScope() method of the ActorImpl class as follows:

```
public boolean removeScope( ScopeImpl aScope )
{
    return focus.remove( aScope );
}
```

As with any object that conforms to the Collection interface, the iterator for an instance of HashSet can be obtained via the iterator() method. The iterator() method is used in TASK any time each of the elements in one of an object's collections needs to accessed in sequential order. For example, the scopes stored in the instance of HashSet representing an actor's focus are iterated over in the destroyImpl() method of the ActorImpl class as follows:

```
...
for ( Iterator i = focus.iterator(); i.hasNext(); )
{
    ScopeImpl scope = (ScopeImpl) i.next();
    scope.unsubscribe( TASKEvent.class, null, this );
    scope.removeObject( this );
}
...
```

One of the more useful aspects of using a HashSet to maintain the collections of objects in TASK is that, like the other concrete collection classes, it implements the Collection interface, and hence can be accessed in a generic way. That is, although the implementation of a class uses a HashSet to store objects in the collection, any access to the collection by clients of the object can be controlled by returning an immutable Collection object rather than the HashSet itself, hence promoting data encapsulation. For example, although the ActorImpl class uses an instance of HashSet to store its focus internally, access to the collection via the focus() method (defined by the Actor interface) returns a Collection as follows:

```
public Collection focus()
{
    return Collections.unmodifiableCollection( focus );
}
```

One of the primary motivators for introducing the Collections API was to provide a set of uniform behaviors and interfaces for groups of objects. The fact that these new classes provide unsynchronized access to their data structures is seen by many as a benefit. Although synchronization allows multiple threads to safely access an object without corrupting the internal state of the object, it comes with a penalty – reduced performance. The new collection classes offer developers greater control over whether access to their collections is synchronized or not. Despite this argument, however, it is a good idea to always make classes threadsafe. This can be accomplished for unsynchronized classes by using static methods offered by the `Collections` class that take an arbitrary collection and turn it into a synchronized collection. Unfortunately, the resulting collections do not correctly synchronize access to methods that involve iteration, so blocks of code that use iterators must be synchronized explicitly using the `synchronized` statement. [Oaks 1998]

In TASK, instances of the unsynchronized `HashSet` class are used instead of instances of the synchronized `Vector` class to represent collections of objects. In order to guarantee mutual exclusion of changes to these collections, the code for the `ActorImpl` constructor presented above must be modified as follows:

```
...
focus = Collections.synchronizedSet( new HashSet() );
...
```

Similarly, due to the unsynchronized nature of methods involving iteration, the code for the `destroyImpl()` method of the `ActorImpl` class presented above must be modified by adding a `synchronized` statement as follows:

```
...
synchronized ( focus )
{
    for ( Iterator i = focus.iterator(); i.hasNext(); )
    {
        ScopeImpl scope = (ScopeImpl) i.next();
        scope.unsubscribe( TASKEvent.class, null, this );
        scope.removeObject( this );
    }
}
...
```

4.2 Implementing Constraints in Java

One method of implementing constraints in a programming language is to use the Design by Contract technique developed by Bertrand Meyer that allows designers and programmers to specify the semantics of a class's interface. At the heart of this technique is the concept of *assertions* – statements that should always be true and can only be false in the event of an error, in which case an exception is raised. Design by Contract uses three kinds of assertions which correspond to the types of constraints described in Chapter 2 – invariants, preconditions, and postconditions. [Fowler 1997]

Using assertions to define the abstract behavior of software elements has several advantages. Component developers, or suppliers, can be assured that software will not be abused as long as correct usage is clearly defined using assertions. In return, components users, or clients, benefit from a precise description of how to use a service and what it will do. Rigorous use of Design by Contract has the potential to also improve the software development cycle as a whole. It represents business rules from the problem domain directly in code thereby confirming that a software system obeys those rules, which leads to traceability. It helps others understand what a class does and builds confidence in the class's performance, which leads to reuseability. It helps uncover code defects earlier by providing a solid foundation for unit testing, which leads to robustness. It allows a module to be considered closed (by specifying its interface fully) while at the same time leaving it open for future changes (assuming the contract is maintained), which leads to extensibility. [Mannion & Phillips 1998]

Invariants, preconditions, and postconditions can easily be implemented in Java by introducing additional classes that evaluate Boolean statements and throw an appropriate exception when an assertion is false. This is accomplished in TASK with three such classes, namely `TASKInvariant`, `TASKPrecondition`, and `TASKPostcondition`. The `TASKAssertionException` class, along with its subclasses, `TASKInvariantException`, `TASKPreconditionException`, and `TASKPostconditionException`, are used to model the associated exceptions. Since the structure of these classes is similar, code for only the `TASKInvariant` class is presented as an example below:

```
public class TASKInvariant
{
    private static boolean enabled = true;

    public static void assert( String description, boolean expression )
        throws TASKInvariantException
    {
        if ( enabled )
            if ( ! expression )
                throw new TASKInvariantException( description );
        return;
    }

    public static void assert( boolean expression )
        throws TASKInvariantException
    {
        if ( enabled )
            if ( ! expression )
                throw new TASKInvariantException();
        return;
    }

    public static void setEnabled( boolean isEnabled )
    {
        enabled = isEnabled;
        return;
    }
}
```

Enforcement of invariants in TASK can be turned on or off by sending the `setEnabled()` message to the `TASKInvariant` class with a Boolean argument. Operations in TASK for which invariants are to be checked simply send the `assert()` message to the `TASKInvariant` class with a Boolean statement and an optional description as arguments. For example, the invariant for the `KeyImpl` class, corresponding to the OCL description provided in Section 2.2.4 of Chapter 2, is checked at the beginning and end of the `describe()` method as follows:

```
...
TASKInvariant.assert( this.invariant() );
...
```

where the implementation of the `invariant()` method of the `KeyImpl` class looks like:

```
public boolean invariant()
{
    if ( ! holders.contains( owner ) )
        return false;
    for ( Iterator i = holders.iterator(); i.hasNext(); )
        if ( ! ( (ActorImpl) i.next() ).getKeys().contains( this ) )
            return false;
    return super.invariant();
}
```

Enforcement of preconditions in TASK can be turned on or off by sending the `setEnabled()` message to the `TASKPrecondition` class with a Boolean argument. Operations in TASK for which preconditions are to be checked simply send the `assert()` message to the `TASKPrecondition` class with a Boolean statement and an optional description as arguments. For example, the preconditions for the `describe()` method of the `KeyImpl` class, corresponding to the OCL description of the `describe` action provided in Section 2.2.4 of Chapter 2, are checked at the beginning of the method as follows:

```
...
TASKPrecondition.assert( owner.equals( requester ) );
TASKPrecondition.assert( ! description.equals( aDescription ) );
...
```

Enforcement of postconditions in TASK can be turned on or off by sending the `setEnabled()` message to the `TASKPostcondition` class with a Boolean argument. Operations in TASK for which postconditions are to be checked simply send the `assert()` message to the `TASKPostcondition` class with a Boolean statement and an optional description as arguments. For example, the postcondition for the `describe()` method of the `KeyImpl` class, corresponding to the OCL description of the `describe` action provided in Section 2.2.4 of Chapter 2, is checked at the end of the method as follows:

```
...
TASKPostcondition.assert( description.equals( aDescription ) );
...
```

Within the context of Design by Contract, interfaces represent a contract between the client and the supplier of a service, the conditions of which are specified by assertions. Through its direct support for interfaces via the `interface` construct, the Java language has no doubt contributed to the growing popularity of the principal of programming to interfaces in recent years. By realizing an interface, a class provides an implementation for the methods it defines; this separation of declaration and implementation forces software developers to think in terms of interfaces, which in turn leads to greater flexibility and ease of reuse. Unfortunately, however, the `interface` construct alone cannot provide a complete specification for an interface because, although it places a syntactical constraint on the signature of a method, it cannot enforce the semantics of the interface. [Mannion & Phillips 1998]

One way of ensuring that both the syntax and the semantics of an interface are consistent is a technique referred to as *down-calling*, also known as the Template Method design pattern (see

[Gamma et al. 1995]). This technique addresses problems that polymorphic methods, or methods for which derived classes can provide a different implementation than their ancestors, present with respect to preserving the semantics of an interface. In particular, derived classes may fail to correctly re-implement the assertions in the method, peer derivations of a class may attempt to enforce different assertions for the same method, or overridden methods may omit the check for assertions entirely. The down-calling approach resolves these issues using two types of methods – interface and implementation. *Interface methods* are publicly accessible methods that can be directly invoked but cannot be overridden; this can be accomplished using the `final` keyword in Java. *Implementation methods*, on the other hand, are not publicly accessible, but are restricted to the inheritance hierarchy and can be overridden to provide the implementation for an interface method. Using this mechanism, interface methods manage the enforcement of assertions whereas implementation methods provide a suitable conforming implementation. [Payne 1997]

The down-calling technique is used in TASK to help ensure that the assertions on the actions are preserved when the framework is extended. Just as each kind of object has an interface and a corresponding implementation class, each action has an interface method and a corresponding implementation method. For example, the interface for keys is `Key`, and the corresponding implementation class is `KeyImpl`. Similarly, the interface method for the `describe` action on keys, as described in Section 2.2.4 of Chapter 2, is `describe()`, and the associated implementation method is `describeImpl()`. The `describe()` method is implemented as follows:

```
public final void describe( Actor aRequester, String aDescription )
    throws TASKException
{
    ActorImpl requester = (ActorImpl) aRequester;

    TASKInvariant.assert( this.invariant() );

    TASKPrecondition.assert( owner.equals( requester ) );
    TASKPrecondition.assert( ! description.equals( aDescription ) );

    this.describeImpl( requester, aDescription );

    TASKPostcondition.assert( description.equals( aDescription ) );

    TASKInvariant.assert( this.invariant() );

    return;
}
```

and the `describeImpl()` method (overridden from `TASKObjectImpl`) looks like:

```
protected void describeImpl( ActorImpl aRequester, String aDescription )
    throws TASKException
{
    this.setDescription( aDescription );

    // send event notification

    return;
}
```

4.3 Remote Objects in Java

Various implementation alternatives exist for client/server applications in the world of network-centric computing. Among these distributed object technologies are CORBA/IIOP, DCOM, RMI, Voyager, HTTP/CGI, and sockets. This section will explore the use of two forms of middleware designed specifically for Java – RMI, from Sun Microsystems, Inc. and Voyager, from ObjectSpace, Inc..

RMI (Remote Method Invocation) is an integrated distributed object model that supports inter-process communication between Java virtual machines. It enables a method of an object in one address space to invoke a method of an object in another address space with the same syntax as a local method call. In addition to allowing the transfer of control between virtual machines and the passing of objects by reference or copy, RMI also supports dynamic class loading and callbacks to applets. RMI is a core part of the Java programming language which all licensees are required to support. [Morrison 1997]

Much like RMI, Voyager is a full-featured, intuitive object request broker that was designed to provide a Java-centric computing platform. In addition to supporting dynamic class loading and callbacks to applets, among other RMI features, Voyager supports mobile objects and autonomous agents and also includes services for persistence, scalable group communication, and federated directories. [ObjectSpace 1997]

In fact, two different implementations of TASK were actually developed, in an effort to explore the practical differences between RMI and Voyager. In the following sections, several aspects of these two technologies are compared in terms of how they are used in TASK.

4.3.1 Remote-Enabling a Class

Remote-enabling a class using RMI requires several steps. First, a remote interface must be defined, which specifies the signature for every method that is to be invoked remotely. This interface must either directly or indirectly extend the `Remote` interface, and each of its methods must declare that they throw the `RemoteException` exception. Next, the class must be made to realize the remote interface and provide an implementation for each of its methods. The class must also either directly or indirectly extend (inherit from) the `UnicastRemoteObject` class or provide its own implementation of built-in remote object behavior. Next, the class must be compiled, and then stubs and skeletons for the class must be generated using the RMI post-compiler, `rmic`. Skeletons are server-side references to remote objects whereas stubs represent client proxies for remote objects that reside on the server. Finally, the resulting stub class files, along with the remote interface class files, must be placed on the client to be used in implementing an application that uses the remote objects. This can either be done manually, by copying the class files to the client machine, or automated using a technique called dynamic class loading, which transfers classes from the server to the client on an as-needed basis via an HTTP process on the server machine.

In the RMI implementation of TASK, the `TASKObject` interface directly extends the `Remote` interface, and hence it and each of its sub-interfaces is implicitly a remote interface. Classes that realize these interfaces also extend the `TASKObjectImpl` class, a direct subclass of `UnicastRemoteObject`, and hence inherit default remote object behavior. The `TASKObject` interface, along with its remote method signatures, is defined as follows:

```
public interface TASKObject extends Remote, TASKPublisher
{
    public Collection actions() throws RemoteException;

    public void addLock( TASKLock aLock ) throws RemoteException;

    public void describe( Actor aRequester, String aDescription )
        throws RemoteException, TASKException;

    public String description() throws RemoteException;

    public void destroy( Actor aRequester )
        throws RemoteException, TASKException;

    public String getDescription() throws RemoteException;

    public Set getLocks() throws RemoteException;

    public String getName() throws RemoteException;

    public boolean isActor() throws RemoteException;

    public boolean isKey() throws RemoteException;

    public boolean isScope() throws RemoteException;

    public boolean isTool() throws RemoteException;

    public String name() throws RemoteException;

    public boolean removeLock( TASKLock aLock ) throws RemoteException;

    public void setDescription( String aDescription ) throws RemoteException;

    public void setName( String aName ) throws RemoteException;
}
```

A similar strategy can also be used to remote-enable classes using Voyager. However, remote interfaces must directly or indirectly extend the `IRemote` interface rather than the `Remote` interface. Unlike RMI, methods that are to be invoked remotely need not declare that they throw a remote exception, and classes that realize the remote interface need not extend any specific class to inherit remote object behavior. Instead of requiring a developer to manually generate stubs and skeletons, Voyager automatically generates client proxies for remote objects on a dynamic, as-needed basis. Client applications need only have access to the remote interface class files, and Voyager takes care of the rest. As with RMI, these class files can be physically copied to the client machine, or dynamically transferred using a remote class-loading mechanism that is built into the Voyager server (described later).

In the Voyager implementation of TASK, the `TASKObject` interface directly extends the `IRemote` interface, and hence it and each of its sub-interfaces is implicitly a remote interface. Classes that realize these interfaces automatically become remote objects as a result of Voyager's proxy generation mechanism. The `TASKObject` interface, along with its remote method signatures, is defined as follows:

```
public interface TASKObject extends IRemote, TASKPublisher
{
    public Collection actions();

    public void describe( Actor aRequester, String aDescription )
        throws TASKException;

    public String description();

    public void destroy( Actor aRequester ) throws TASKException;

    public boolean isActor();

    public boolean isKey();

    public boolean isScope();

    public boolean isTool();

    public String name();
}
```

4.3.2 Exporting a Named Object

Once a class has been remote-enabled, instances of the class must be made available to client applications by exporting named references to them. RMI includes an object registry service that can be used to bind a remote object to a name, thus exporting the object for use by remote clients. Once the remote registry has been started on the server either from the command line or programmatically, an object can be bound to a URL-based name by sending the `rebind()` message to the `Naming` class.

In the RMI implementation of TASK, the TASK class is introduced to centralize the starting of the registry as well as the binding and unbinding of remote objects. For example, the `startup()` method starts the RMI registry on a specific network port as follows:

```
private static void startup()
{
    try
    {
        ...
        LocateRegistry.createRegistry( PORT );
    }
    catch ( Exception e )
    {
        ...
    }
    return;
}
```

Each remote-enabled class in the RMI implementation of TASK exports its instances by binding them to names in the object registry in one of its constructors. This is accomplished using the `create()` method of the TASK class, which binds remote objects by sending the `rebind()` message to the `Naming` class with the desired name and the object as arguments, as follows:

```
public static void create( TASKObjectImpl object )
    throws TASKException
{
    try
    {
        Naming.rebind( URL + object.getName(), object );
    }
    catch ( Exception e )
    {
        throw new TASKException( e );
    }
    ENVIRONMENT.add( object );
    return;
}
```

Voyager also provides an integrated naming service that can be used to export references to remote objects. This naming service is part of the Voyager server, which must be started on both the server and client machines before objects can send and receive messages between them. Once the Voyager server process has been started either from the command line or programmatically, an object can be bound to a URL-based name by sending the `rebind()` message to the `Namespace` class.

The Voyager implementation of TASK also uses one class to centralize the starting of the Voyager server as well as the binding and unbinding of remote objects. For example, the `startup()` method of the TASK class starts the Voyager server on a specific network port as follows:

```
private static void startup()
{
    try
    {
        ...
        Voyager.startup( PORT );
    }
    catch ( Exception e )
    {
        ...
    }
    return;
}
```

Each remote-enabled class in the Voyager implementation of TASK exports its instances by binding them to names in the naming service in one of its constructors. This is accomplished using the `create()` method of the TASK class, which binds remote objects by sending the `rebind()` message to the `Namespace` class with the desired name and the object as arguments, as follows:

```
public static void create( TASKObjectImpl object )
    throws TASKException
{
    try
    {
        Namespace.rebind( URL + object.getName(), object );
    }
    catch ( Exception e )
    {
        throw new TASKException( e );
    }
    ENVIRONMENT.add( object );
    return;
}
```

4.3.3 Obtaining a Reference to a Remote Object

Assuming a client application has access to the interface and stub class files for a remote class, it can obtain a remote reference to instances of the class by looking up its name in the remote registry. This can be accomplished by sending the `lookup()` message to the `Naming` class with the object's URL-based name as an argument, and casting the result to the expected type. Doing this, of course, may result in exceptions such as `NotBoundException`, `MalformedURLException`, and `RemoteException`, all of which must be explicitly caught or re-thrown. For example, a client application could obtain references to a remote scope and actor in the RMI implementation of TASK as follows:

```
try
{
    Scope scope = (Scope) Naming.lookup( "//khussey:7000/task" );
    Actor actor = (Actor) Naming.lookup( "//khussey:7000/kenn" );
}
catch ( Exception e )
{
    ...
}
```

As with RMI, Voyager provides a means to look up remote objects by name using its naming service. A reference to a remote object can be obtained by sending the `lookup()` message to the `Namespace` class with the object's URL-based name as an argument, and casting the result to the expected type. Doing this may result in exceptions such as `NamespaceException`, which must be explicitly caught or re-thrown. For example, a client application could obtain references to a remote scope and actor in the Voyager implementation of TASK as follows:

```
try
{
    Scope scope = (Scope) Namespace.lookup( "//khussey:8000/task" );
    Actor actor = (Actor) Namespace.lookup( "//khussey:8000/kenn" );
}
catch ( Exception e )
{
    ...
}
```

4.3.4 Invoking a Method Remotely

Invoking a method on a reference to a remote object using RMI is essentially no different from invoking a method on a normal object in Java except that there is a possibility that the `RemoteException` exception may be thrown. RMI substitutes a skeleton or stub for every method argument, parameter, or return value that is an instance of a remote-enabled class. One unfortunate limitation of RMI is that there is no way to get a reference to the implementation object from its skeleton proxy. Consequently, any server-side method invoked on an object that is passed as a remote parameter must still be declared as part of a remote interface. For this reason, remote interfaces in the RMI implementation of TASK must, in general, define more methods than the corresponding interfaces in the Voyager implementation. A client application could remotely invoke the `enter()` method on its reference to the remote scope in the RMI implementation of TASK as follows:

```
try
{
    scope.enter( actor );
}
catch ( Exception e )
{
    ...
}
```

Using Voyager, invoking a method on a reference to a remote object is essentially no different from invoking a method on a normal object in Java except that there is a possibility that the `RuntimeRemoteException` exception may be thrown. Voyager substitutes a proxy object for all method arguments, parameters, and return values that are instances of classes which implement a remote interface. The implementation object can be obtained from its server-side proxy in Voyager by sending it the `getLocal()` message and casting the result to the expected type. Consequently, server-side methods that won't be invoked remotely need not be declared as part of a remote interface. For this reason, Voyager provides better support than RMI does for encapsulation in the sense that only high-level, interface methods must be exposed in remote interfaces while low-level implementation methods can be hidden in the classes that realize these interfaces. A client application could remotely invoke the `enter()` method on its reference to the remote scope in the Voyager implementation of TASK as follows:

```
try
{
    scope.enter( actor );
}
catch ( Exception e )
{
    ...
}
```

4.4 Object Serialization in Java

Persistence is the ability of an object to save its state so that it can be restored and used at a later time. Persistence can be implemented in Java using a technique called *object serialization*, which converts data structures into a common data stream that is independent of processor or operating system. Any class can take advantage of this mechanism simply by implementing `Serializable`, an interface that actually has no methods but is used by Java as a marker to determine whether an object can be serialized. By realizing this interface, a class implicitly inherits the default algorithm for converting an object to and from a data stream. [Wong 1998]

All objects in TASK are implicitly serializable because the `TASKObjectImpl` class implements the `Serializable` interface. As a result, the server can easily store the state of the environment by asking each of the objects in TASK to write themselves to an output stream. This is done indirectly by writing out a collection containing the objects in the `save()` method of the TASK class, as follows:

```
public static void save()
    throws TASKException
{
    try
    {
        FileOutputStream fos = new FileOutputStream( "task.dat" );
        ObjectOutputStream oos = new ObjectOutputStream( fos );
        oos.writeObject( ENVIRONMENT );
        oos.close();
    }
    catch ( Exception e )
    {
        throw new TASKException( e );
    }
    return;
}
```

Similarly, the server can easily restore the state of the TASK environment by asking objects to read themselves from an input stream. This done indirectly by reading in a collection containing the objects in the `restore()` method of the TASK class, as follows:

```
private static void restore()
    throws TASKException
{
    try
    {
        FileInputStream fis = new FileInputStream( "task.dat" );
        ObjectInputStream ois = new ObjectInputStream( fis );
        ENVIRONMENT = (Set) ois.readObject();
        ois.close();
        ...
    }
    catch ( Exception e )
    {
        throw new TASKException( e );
    }
}
```

Although the default algorithm for object serialization in Java works for the majority of objects, there are some cases where this mechanism is not sufficient. Fields in a class can be excluded from the default serialization mechanism by declaring them transient. Data that still needs to be made persistent, however, can be serialized in a customized way by overriding the `writeObject()` and `readObject()` methods. Combining use of the `transient` keyword with implementation of the `writeObject()` and `readObject()` methods retains the ease of use afforded by the `Serializable` interface while at the same time providing the flexibility of application-specific serialization. [Wong 1998]

In TASK, the `subscriptions` attribute of the `TASKContextImpl` class is declared transient so that any subscriptions involving instances of classes that implement the `TASKSubscriber` interface but do not inherit from `TASKObjectImpl` class are not made persistent. In order to ensure that the remaining subscriptions can still be stored to an output stream, however, the TASKContextImpl class overrides the writeObject() method, as follows:

```
private void writeObject( ObjectOutputStream oos )
    throws IOException
{
    oos.defaultWriteObject();
    Set s = Collections.synchronizedSet( new HashSet() );
    Synchronized ( subscriptions ) {
        for ( Iterator i = subscriptions.iterator(); i.hasNext(); )
        {
            TASKSubscription ts = (TASKSubscription) i.next();
            if ( TASKObjectImpl.class.isInstance( ts.getSubscriber() ) )
                s.add( ts );
        }
    }
    oos.writeObject( s );
    return;
}
```

Similarly, in order to read serialized subscriptions from an input stream when instances of TASKContextImpl are restored, the readObject() method must be overridden as follows:

```
private void readObject( ObjectInputStream ois )
    throws IOException, ClassNotFoundException
{
    ois.defaultReadObject();
    subscriptions = (Set) ois.readObject();
    return;
}
```

5 CONCLUSION

This thesis has described TASK, a general framework for collaborative workspaces. In particular, it has explored the conceptual model, design specification, and implementation of TASK, in an effort to demonstrate how it supports several desirable features of groupware to support collaborative work. In conclusion, this chapter briefly reviews the specific aspects of TASK that support these features and thus facilitate the development of useful collaborative environments.

Frames of reference for collaborative activities are represented by scope objects in TASK. Unlike the simplistic notion of rooms employed by traditional environments, scopes provide a context for collaboration that transcends the limitations of a spatial metaphor. TASK allows actors to be part of more than one scope concurrently, which means that participation in activities is not limited on a per-context basis.

Communication between and within these frames of reference is accomplished by representing users as actors in TASK. Interactions between these actors and other objects can be direct or indirect, synchronous or asynchronous, and one-to-one or one-to-many. Awareness in TASK is facilitated through application of the Event Notifier design pattern. Using this mechanism, the execution of actions on objects is communicated as events to other objects in the environment, allowing users to be aware of the activities of other users.

Tools to support the activities performed within these frames of reference are represented by tool objects in TASK. The primary means by which activities are performed in TASK, tools provide a simple foundation that can be extended and adapted to support a more complete range of activities. The ability to integrate new tools and actions makes TASK an ideal framework for developing environments to support collaborative work.

BIBLIOGRAPHY

Booch, G., Rumbaugh, J., and Jacobson, I.. 1999. *The Unified Modeling Language User Guide.* Addison-Wesley Longman, Inc., Reading, MA.

Curtis, P.. 1993. LambdaMOO Programmer's Manual (available from parcftp.xerox.com, directory pub/MOO/papers).

Curtis, P., and Nichols, D. A.. 1993. "MUDs Grow Up: Social Virtual Reality in the Real World". In *Third International Conference on Cyberspace*, Austin, TX (available from parcftp.xerox.com, directory pub/MOO/papers).

Evard, R.. 1993. "Collaborative Networked Communication: MUDs as Systems Tools". In *Proceedings of the Seventh System Administration Conference (LISA VII)*, Montery, CA (available from parcftp.xerox.com, directory pub/MOO/papers).

Fowler, M. with Scott, K.. 1997. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley Longman, Inc., Reading, MA.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J.. 1995. *Design Patterns. Elements of Reusable Object Oriented Software.* Addison-Wesley Publishing Company, Inc., Reading, MA.

Gupta, S., Hartkopf, J., and Ramaswamy, S.. 1998. "Event Notifier: A Pattern for Event Notification". *Java Report*, 3(7):19-36.

Hunt, J.. 1999. "The Collection API". *Java Report*, 4(4):17-32.

Hussey, K.. 1996. *Design and Implementation of a MUD Framework in Smalltalk.* Undergraduate honours thesis, Acadia University, NS, Canada.

Hussey, K. and Müldner, T.. 1998. "Virtual Collaborative Working Environments". In *Proceedings of ED-MEDIA 98 - World Conference on Educational Multimedia and Hypermedia*, Freiburg, Germany.

Hussey, K. and Tomek, I.. 1996. "Support for Cooperation in Smalltalk". In *Proceedings of ED-TELECOM 96 - World Conference on Educational Telecommunications*, Boston, MA.

Mannion, M. and Phillips, R.. 1998. "Prevention is Better Than a Cure". *Java Report*, 3(9):23-36.

Morrison, M. et al.. 1997. *Java 1.1 Third Edition.* Sams.net Publishing, Indianapolis, IN.

Oaks, S.. 1998. "The burden of synchronization: Hashtable vs. HashMap". *Java Report*, 3(8):78-80.

ObjectSpace, Inc.. 1997. "Voyager and RMI Comparison". ObjectSpace, Inc., Dallas, TX.

Payne, J., Alexander, R., and Hutchinson, C.. 1997. "Design-for-Testability for Object-Oriented Software: Techniques for increasing software testability". *Object Magazine*, 7(5):35-43.

Rodden, T.. 1996. "Populating the Application: A Model of Awareness for Cooperative Applications". In *Proceedings of ACM CSCW'96 Conference on Computer-Supported Cooperative Work*, Cambridge, MA.

Warmer, J. and Kleppe, A.. 1999a. "OCL: The Constraint Language of the UML". *The Journal of Object-Oriented Programming*, 12(2):10-13,28.

Warmer, J. and Kleppe, A.. 1999b. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley Longman, Inc., Reading, MA.

Wong, H.. 1998. "What is Java bean persistence?". *Java Report*, 3(3):70-72.

APPENDIX A: OBJECT REFERENCE

Actor

An individual who collaborates, or coacts, within TASK.

Attributes

context

The frame of reference for actions invoked on the actor.

description

A textual description of the actor.

focus

The scopes which the actor has entered.

keys

The keys which have been granted to the actor.

name

The unique textual identifier for the actor.

objects

The objects (tools) for which the actor is a frame of reference.

Actions

create

Allows the requester to create a new actor, scope, tool, or key.

describe

Allows the requester to change the description of the actor.

destroy

Allows the requester to destroy the actor.

focalize

Allows the requester to change the context for the actor.

lock

Allows the requester to lock one of the actor's actions with a key.

unlock

Allows the requester to unlock one of the actor's actions with a key.

Key

A mechanism of limiting the behaviors of actors within TASK.

Attributes

description

A textual description of the key.

holders

The actors to which the key has been granted.

name

The unique textual identifier for the key.

owner

The actor that has authority to grant or revoke the key.

Actions

describe

Allows the requester to change the description of the key.

destroy

Allows the requester to destroy the key.

grant

Allows the requester to grant the key to an actor.

revoke

Allows the requester to revoke the key from an actor.

Scope

A frame of reference for the actions that actors engage in as they collaborate in TASK.

Attributes

context

The frame of reference for actions invoked on the scope.

description

A textual description of the scope.

name

The unique textual identifier for the scope.

objects

The objects (actors, scopes, and tools) for which the scope is a frame of reference.

Actions

describe

Allows the requester to change the description of the scope.

destroy

Allows the requester to destroy the scope.

enter

Allows the requester to enter the scope.

exit

Allows the requester to exit the scope.

lock

Allows the requester to lock one of the scope's actions with a key.

unlock

Allows the requester to unlock one of the scope's actions with a key.

Tool

A means by which actions are performed in TASK.

Attributes

context

The frame of reference for actions invoked on the tool.

description

A textual description of the tool.

name

The unique textual identifier for the tool.

Actions

describe

Allows the requester to change the description of the tool.

destroy

Allows the requester to destroy the tool.

drop

Allows the requester to put the tool down.

lock

Allows the requester to lock one of the tool's actions with a key.

take

Allows the requester to pick the tool up.

unlock

Allows the requester to unlock one of the tool's actions with a key.