A Prototyping Technique to Verify Requirements

Shih-Chien Chou
Department of Computer Science and Information Engineering
National Dong Hwa University, Hualien 974, Taiwan
E-mail: scchou@mail.ndhu.edu.tw

ABSTRACT

This article proposes a prototyping technique to verify requirements. The technique is composed of a model to represent requirements and an executable language to create prototypes based on requirements. The requirement model is composed of use case diagram, activity diagram, and class diagram. The language models those diagrams as a prototype, which can be executed to verify requirements before they are modeled into a specification. Since requirement errors will propagate to the corresponding specification, correcting requirement errors reduces possible specification errors.

1. INTRODUCTION

The prototyping technique facilitates verifying specifications [1]. That is, after a specification has been produced, a prototype can be created based on the specification. The prototype can then be executed to verify the specification.

According to our experiences in object-oriented analysis (OOA) [2-3], we feel that applying the prototyping technique to verify specifications may be relatively inefficient. A better approach is using that technique to verify requirements before they are modeled into a specification, because requirement errors will propagate to the corresponding specification. Since much time is needed in analyzing requirements and modeling them into a specification, correcting requirement errors consumes much less resources than correcting specification errors.

We first describe a typical requirement analysis process and then propose our technique to verify requirements. Generally, requirement analysis starts with requirement capturing. Requirements are then modeled into a specification. During requirement capturing, the *requirement workers*, including customers, analysts, end users, domain experts, and so on, identify requirements according to their perspectives. The requirements identified are then analyzed and integrated in a meeting. The meeting results (i.e., the requirements after integration) are then represented in a model (perhaps the same model for the specification, or just a relatively structured language). The requirement workers then verify the meeting results. If errors or new requirements are identified, another meeting is needed. Therefore, multiple meetings may be necessary before the requirements become stable. The stable requirements are then modeled into a specification.

The above process reveals that requirement workers should verify requirements and identify requirement errors. Requirements should thus be represented in a model that can be easily understood by every requirement worker. Since the knowledge background of requirement workers may diverge dramatically, identifying a model that can be easily understood by all the workers may be difficult. Therefore, it is possible that some of the workers have difficulty in understanding requirements. This may result in unidentified requirement errors, which will propagate to the specification. To reduce requirement errors, we have applied the prototyping technique to verify requirements. That is, we create a prototype based on the requirements captured after each meeting. The prototype is then executed to verify the requirements. The prototype will evolve as the requirements are adjusted in the succeeding meetings. When the requirements become stable and the specification has been produced, the prototype becomes a prototype based on the specification (i.e., a prototype in the traditional approach). Therefore, our technique can also be applied in a traditional prototyping approach.

Similar to most prototyping techniques [4-10], our technique uses an executable language to model prototypes. The language is designed for our requirement capturing technique [3], in which the use case diagram, activity diagram, and class diagram are used to represent requirements. In the following text, we first describe our requirement model. We then describe our prototyping language. To facilitate understanding the description, we use a simplified car rental system as an example throughout this article.

2. THE MODEL

As stated in the unified software development process [11], use cases and domain objects can be identified during requirement capturing. We use a use case diagram to depict use cases and their relationships, and a class diagram to depict domain objects and their relationships. Moreover, to facilitate understanding use cases, we use an activity diagram to represent the detailed activities of each use case. In this section, we first describe an example used throughout this article. We then describe our requirement model, including the use case diagram, the activity diagram, and the class diagram. Note that this article does not describe the requirement capturing process.

2.1 The Example

We use a simplified car rental system as an example throughout this article.

Requirements of the system are described below:

> *The car rental system manages the status of cars and renters. When a renter rents a car, a rental record is created. A renter may be a member or not. A member can rent a car if he/she has no bad record. A non-member should apply for becoming a member or mortgage something valuable before he/she rents a car. Old cars can be removed and new cars can be added.*
>
> *When a rented car is returned, the car should be checked. If the car is damaged, the renter should compensate for that.*
>
> *The manager of the system should periodically check car status. A car with an age more than ten years should be removed. The manager should also periodically check rental records. A bad record should be created for a member who does not return a car in time.*
>
> *The system should also handle exceptions, which are events that cannot be controlled regularly. For example, a renter may decide not to rent a car during the handling of renting the car. In this case, the system should recover the rental. Since a renter may change his/her mind anytime, un-renting a car cannot be controlled regularly. The un-renting should thus be managed as an exception.*

From the requirements, we can identify two actors, namely the renter and the manager. Moreover, we can identify the following use cases: rent a car, return a car, check car status, check rental record, damage compensation, add a member, remove a member, add a car, and remove a car. In addition, we can at least identify an exception, namely un-renting a car.

2.2 Use Case Diagram

A use case diagram describes how users use a system. In the unified modeling language (UML) [12], a use case diagram is composed of these components: use cases, actors, and relationships among use cases (see Figures 1(a) through 1(d)). Initially, we used those components in our use case diagram. We then identified that important features such as use case communication and exceptions cannot be modeled. We thus added new notations for our use case diagram.

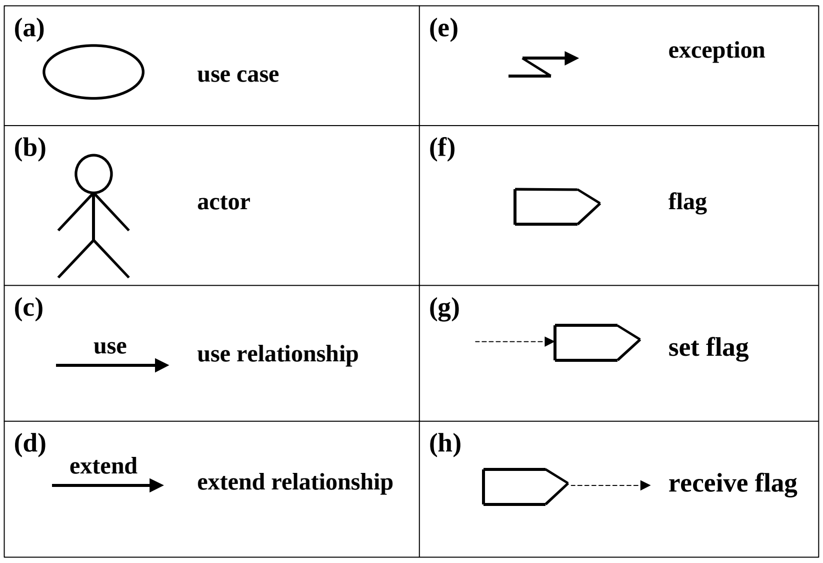


Figure 1. Notations for use case diagram

Figure 1(e) models exceptions. Exception names are associated with the notation. The arrow points to the handler of the exception, which is generally a use case. Figure 2 shows an example of exception modeling, which depicts that the exception “un-rent” may occur anytime when the use case “Rent a car” is executing. If that exception occurs, the use case “Rental recovery” is executed to recover the rental. Here, the use case “Rental recovery” is the handler of the exception.

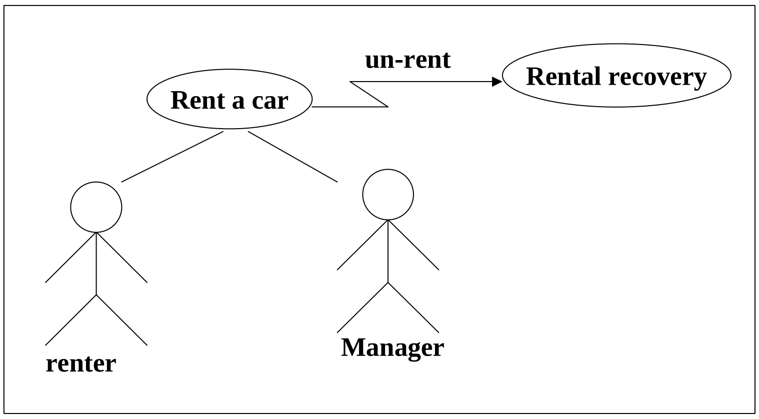


Figure 2. Exception

Flags (Figures 1(f) through 1(h)) model use case communication. For example, in the simplified car rental system, when the use case “Rent a car” is executing (i.e., when a renter rents a car), it should inform the use case “Check car status” to check whether the car is available. The communication between the above two use cases is shown in Figure 3, which depicts that the flag “checkCarF” is set by the use case “Rent a car” and received by “Check car status”. The setting and receiving of flags accomplishes use case communication. When a use case receives a flag, the use case will be executed. Note that a flag may cause only partial activities of the receiver to execute. For example, although the use case “Check car status” checks the status of every car, the flag “checkCarF” causes that use case to check only a specific car (i.e., the car to rent).

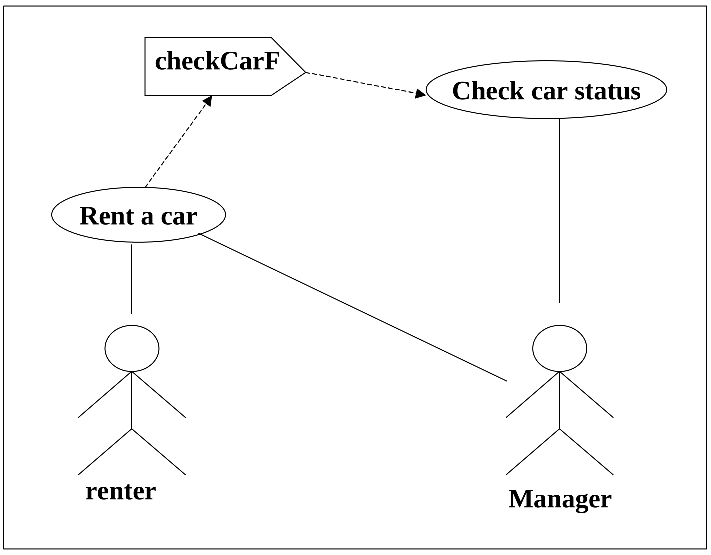


Figure 3. Use case communication

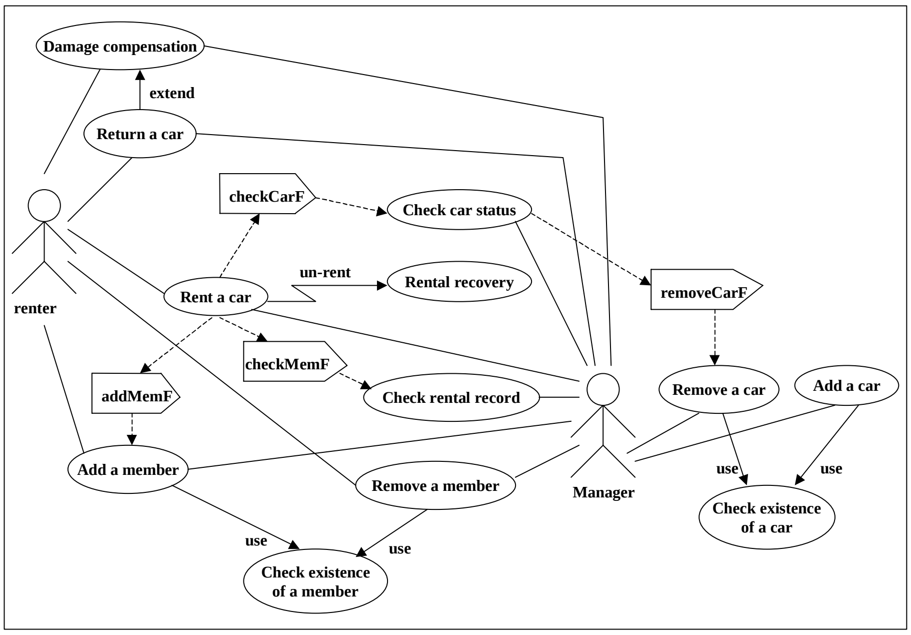


Figure 4. Use case model for the simplified car rental system

Using the notations in Figure 1, the use case diagram of the simplified car rental system is shown in Figure 4. Nine use cases in the figure have been mentioned in section 2.1. As to the use case "Rental recovery", it handles the exception "un-rent". Moreover, the use cases "Check existence of a member" and "Check existence of a car" are used by multiple use cases. The figure also shows use case communication using flags. For example, the use case "Rent a car" communicates with "Add a member” via the flag “addMemF”. Communication details among use cases will be shown in the activity diagrams of use cases.

2.3 Activity Diagram

Activity diagrams describe the detailed activities of use cases. This facilitates further understanding use cases. Notations used in an activity diagram are shown in Figure 5. They are explained below:

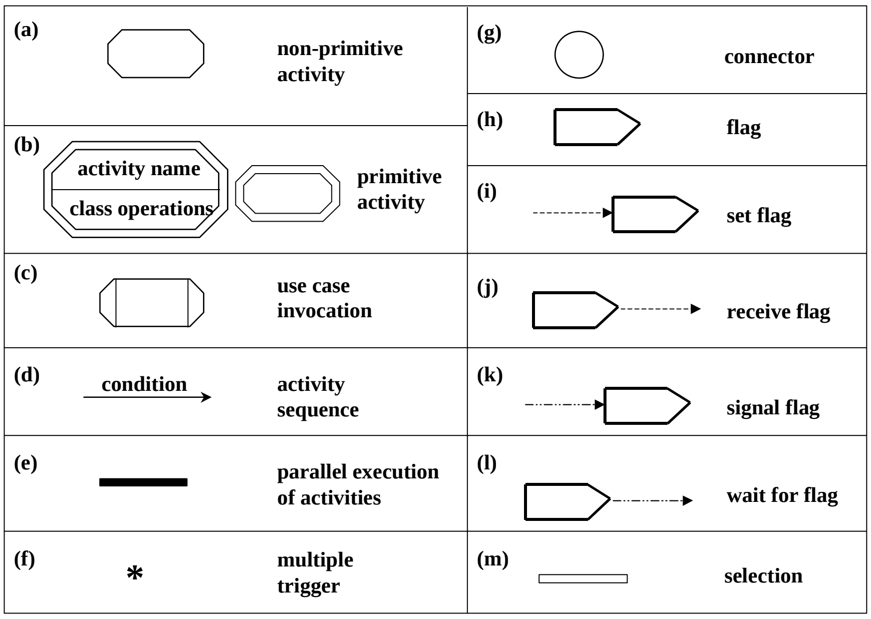


Figure 5. notations for activity diagram

1) Figure 5(a) models non-primitive activities, which are complicated and should be decomposed to improve understandability.
2) Figure 5(b) models primitive activities, which are rather simple and therefore need not be decomposed.

   Some primitive activities can be accomplished by invoking class operations. For example, the activity “Remove a car” can be accomplished by invoking the operation “remove” of the class “carClass”. If a primitive activity can be accomplished by invoking class operations, the left notation in Figure 5(b) is used. The notation is partitioned into two fields. The first field shows the activity name and the second shows class operations. Placing class operations in primitive activities maintains traceability between use cases and classes, with which changing use cases can trace back to the affected classes, and vice versa.

If a primitive activity is not accomplished by invoking class operations, the right notation in Figure 5(b) is used.

3) Figure 5(c) models use case invocation, which is used in the "use" and "extend" relationships among use cases. For example (see Figure 4), with the "extend" relationship between the use cases "Return a car" and "Damage compensation", the activity diagrams for "Return a car" is shown in Figure 6, in which the use case "Damage compensation" is invoked if the car is damaged. As another example, with the "use" relationship between the use cases "Add a car" and "Check existence of a car", the activity diagrams for "Add a car" is shown in Figure 7, in which the use case "Check existence of a car" is invoked.

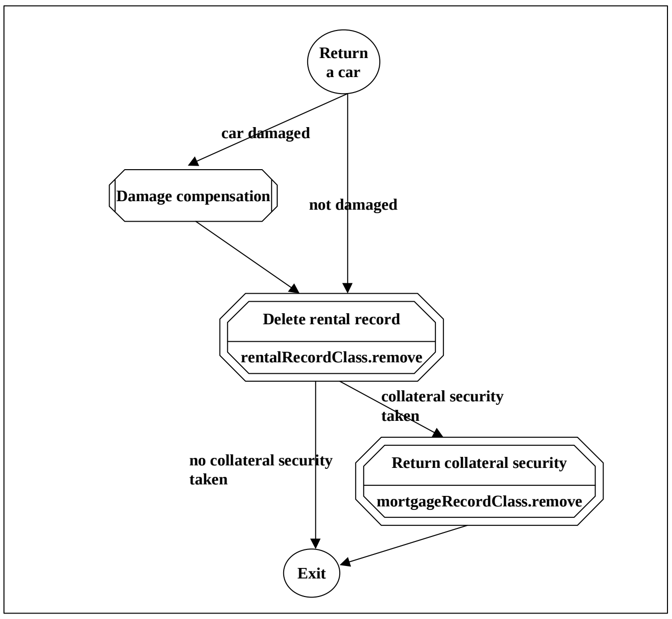

Return a car
car damaged
Damage compensation
not damaged
Delete rental record
rentalRecordClass.remove
collateral security taken
no collateral security taken
Return collateral security
mortgageRecordClass.remove
Exit

Figure 6. Activity diagram of the use case "Return a car"

4) Figure 5(d) models activity sequence. That is, for the activities connected by solid arrows, the successors can be executed only when the predecessors are finished. Conditions can be associated with the notation. For example (see Figure 6), if a car is damaged when it is returned, the use case "Damage compensation" will be invoked before the activity "Delete rental record" executes.

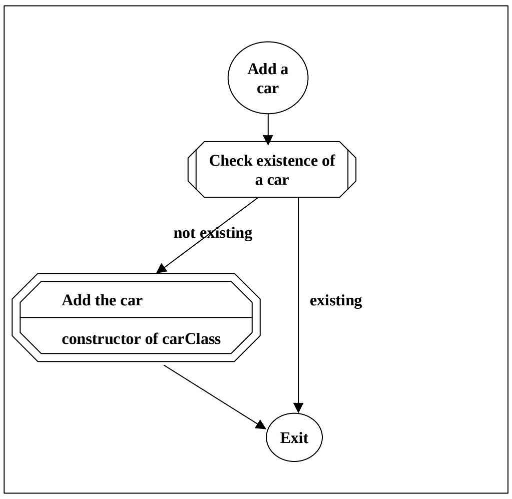


Figure 7. Activity diagram of the use case " Add a car"

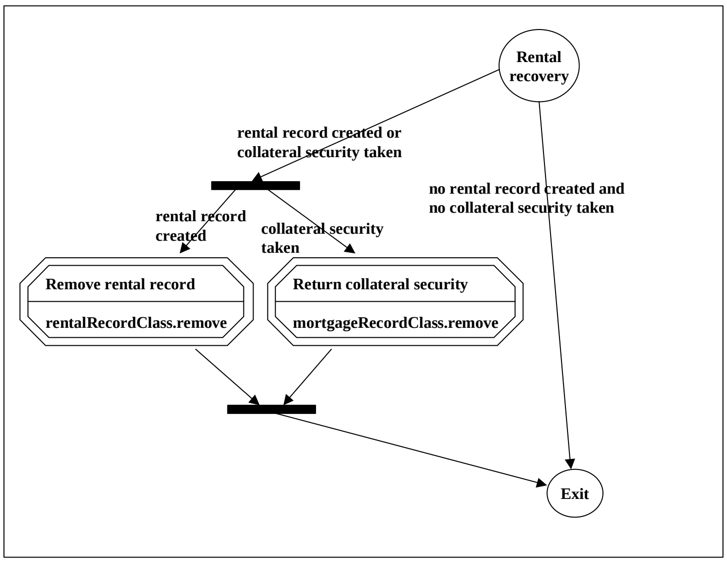

Rental recovery
rental record created or collateral security taken
no rental record created and no collateral security taken
rental record created
collateral security taken
Remove rental record
rentalRecordClass.remove
Return collateral security
mortgageRecordClass.remove
Exit

Figure 8. Activity diagram of the use case "Rental recovery"

5) Figure 5(e) models parallel execution of activities. For example, Figure 8 shows the activity diagram of the use case "Rental recovery". It depicts that if a rental record has been created or collateral security has been taken, then the activities "Remove rental record" and "Return collateral security" can be executed in parallel.

6) Figure 5(f) models multiple triggers. For example, Figure 9 shows the activity

diagram for the use case “Check car status”. Since every car should be checked and the number of cars is unknown, a multiple trigger can be used here. The figure depicts that multiple copies of the activities “Check if the car is older than 10 years” and “Add a car status record” can be performed in parallel.

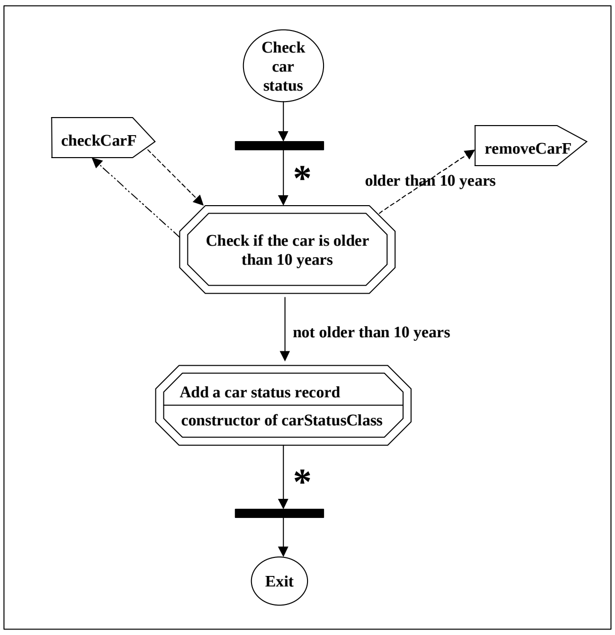


Figure 9. Activity diagram of the use case “Check car status”

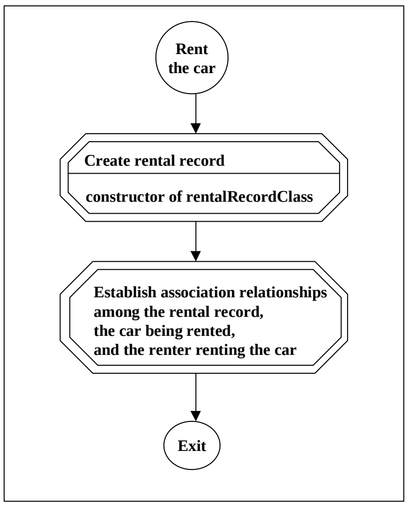

Rent the car
Create rental record
constructor of rentalRecordClass
Establish association relationships among the rental record, the car being rented, and the renter renting the car
Exit

Figure 10. Activity diagram for the activity “Rent the car” in Figure 11

7) Figure 5(g) denotes a *connector*, which shows the name of the use case represented by an activity diagram. The notation also shows the exit of an activity diagram.

For example, the upper connector in Figure 9 depicts that the diagram represents the use case “Check car status”, whereas the lower connector is the exit of the diagram. Connectors also show decomposition relationships among activities. For example, Figure 10 shows the activity diagram obtained by decomposing the activity “Rent the car” in Figure 11. Naming the upper connector as “Rent the car” shows the activity decomposition relationship.

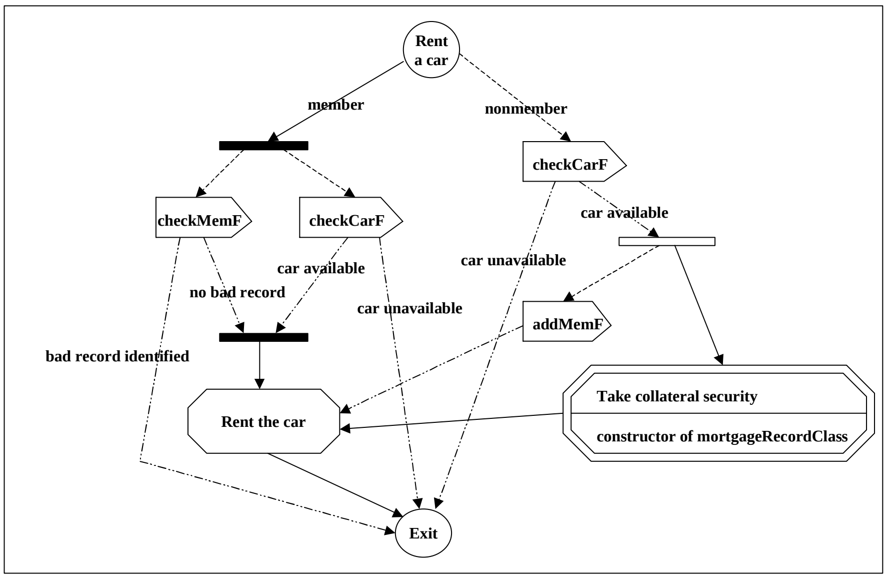


Figure 11. Activity diagram for the use case “Rent a car”

8) Figure 5(h) models flags for use case communication. Although the use case diagram shows the setting and receiving of flags among use cases (see Figure 4), that diagram does not show when and how the flags are set and received. Detailed flag operations are modeled in activity diagrams, in which Figures 5(i) through 5(l) model setting, receiving, signaling, and waiting for flags.

Figure 5(i) models the setting of a flag. The dotted arrow points to the flag to set. The setting of a flag is used by a use case to communicate with other use cases. For example (see Figure 11), the use case “Rent a car” sets the flags “checkCarF”, “checkMemF”, and “addMemF” to communicate with the use cases “Check car status” (Figure 9), “Check rental record” (Figure 12), and “Add a member” (Figure 13). To identify which use case receives a flag, Figure

5(j) is used.

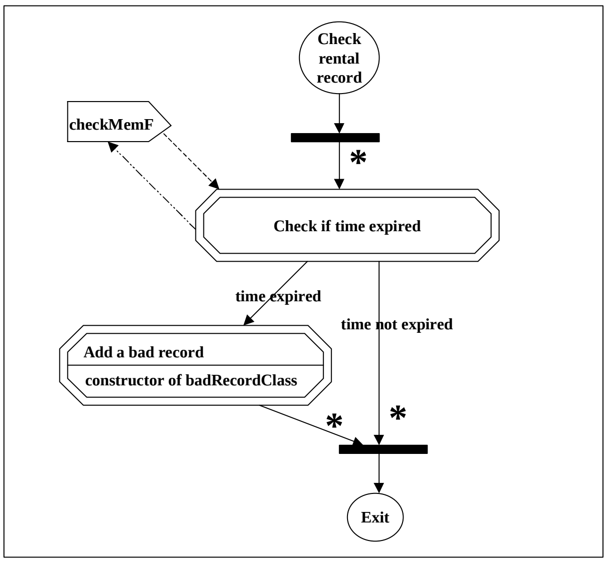


Figure 12. Activity diagram for the use case “Check rental record”

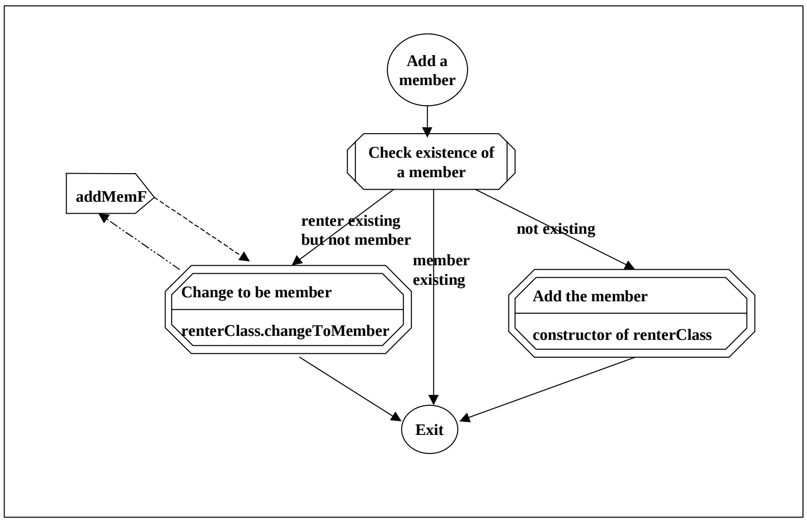

Add a member
Check existence of a member
addMemF
renter existing but not member
not existing
member existing
Change to be member
renterClass.changeToMember
Add the member
constructor of renterClass
Exit

Figure 13. Activity diagram for the use case “Add a member”

Figure 5(j) models the receiving of a flag. The dotted arrow points to the activity that should be executed when the flag is received. For example, when the use case “Rent a car” sets the flag “checkMemF”, the activity “Check if time expired” of the use case “Check rental record” (Figure 12) should be executed. Note that it may be unnecessary to execute the entire use case when a flag is received. For example, when the use case “Check rental record” receives the flag “checkMemF”, it executes only the activity “Check if time expired”.

Generally, when the use case that receives a flag finishes executing the activities required by the flag, the use case should inform the use case that set the flag. This informing is accomplished by signaling a flag, which is modeled by Figure 5(k). The dotted arrow pointed to the flag to signal. For example, Figure 9 depicts that the flag “checkCarF” requires the executing of the activity “Check if the car is older then 10 years”. When the activity is finished, that flag is signaled. A signaled flag will be detected by the use case waiting for the flag.

Waiting for a flag is used when a use case that sets a flag can proceed only when the flag is signaled. That waiting is modeled by Figure 5(l). The dotted arrow points to the activity to proceed when the flag is signaled. For example, in the left side of Figure 11, when the use case “Rent a car” sets the flag “checkCarF”, it waits for the flag to be signaled. When the flag is signaled, the use case proceeds by exiting the use case (in case that the car is unavailable) or proceeds to the synchronization bar.

9) Figure 5(m) models selection. For example, Figure 11 depicts that a non-member can rent a car by either becoming a member or mortgaging something valuable. In the figure, the selection bar depicts that either, but not both, the flag “addMemF” is set (to register the renter as a member) or the activity “Take collateral security” is executed (to mortgage something valuable).

Using the notations in Figure 5, the activity diagrams of some use cases in the car rental system are modeled as those in Figures 6 through 13. We use the use case “Rent a car” (Figure 11) as an example to explain the usage of the notations. The figure depicts that when a renter wants to rent a car, he/she is first checked to see whether he/she is a member. If he/she is a member, two flags, namely “checkMemF” and “checkCarF”, are concurrently set. The former informs the use case “Check rental record” (Figure 12) to check whether there is a bad record associated with the member.

The latter flag informs the use case “Check car status” (Figure 9) to check whether the car is available. If no bad record is associated with the member and the car is available, the renter rents the car. The renting is accomplished by invoking the non-primitive activity “Rent the car” (Figure 10). If the renter is not a member, he/she can apply for becoming a member (i.e., the system sets the flag “addMemF” to inform the use case “Add a member” in Figure 13 to register the member) and then rents the car. If the renter does not want to become a member, he/she should mortgage something valuable before renting the car.

2.4 Class Diagram

The class diagram models classes and their relationships. Notations used in the class diagram are depicted in Figure 14. Figure 14(a) sketches the notations for classes. The left one displays only a class name. It is used when class attributes and operations need not be shown. The right notation is used when class attributes and operations are needed. Figure 14(b) sketches an inheritance relationship, where the super class is drawn on top of its subclasses. Figure 14(c) depicts a composition relationship, where the composite class is next to the diamond shape. Figure 14(d) depicts association relationships, which are those other than the inheritance and composition relationships.

Figure 15 depicts the class diagram of the car rental system. To simplify the figure, class attributes and operations are not shown.

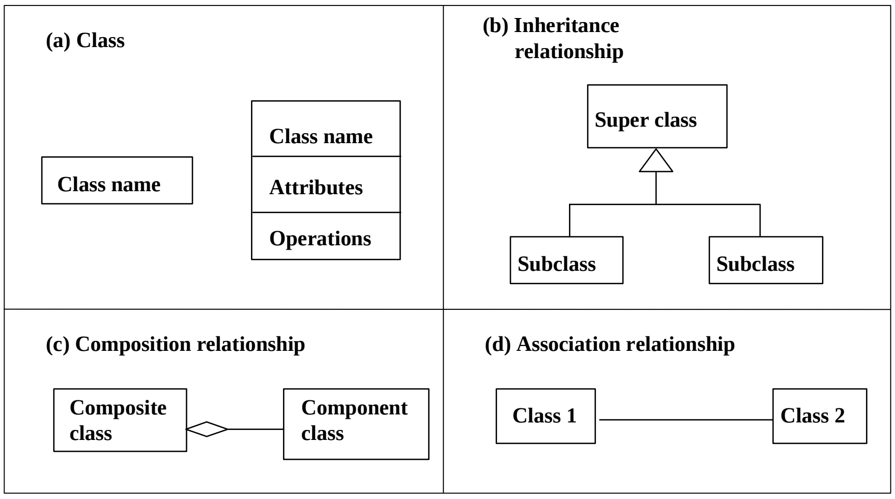


Figure 14. Class diagram notations

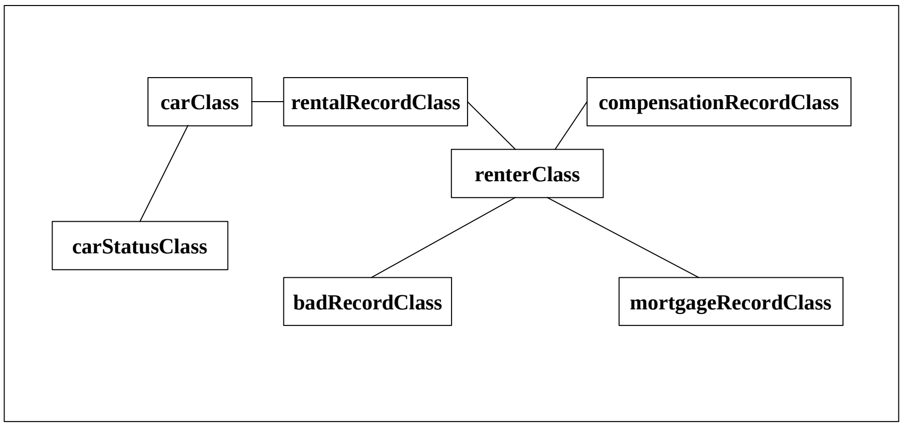


Figure 15. Class diagram for the simplified car rental system

3. THE PROTOTYPING LANGUAGE

We first describe the design philosophy of our prototyping technique and then describe our prototyping language.

3.1 Design philosophy

To create a prototype based on requirements, the use case diagram, activity diagrams, and class diagram are represented in our prototyping language. A prototype should facilitate verifying the functions of a system, because a software system provides functions to fulfill customer requirements. Since the functions of a system in our model are captured by use cases, our technique executes use cases for requirement verification.

When executing a use case, users (either customers or analysts) provide input data. After the execution, output data are produced, which can be compared with the expected output to verify the use case. In verifying a use case, our technique goes a step further to show the state transitions of objects. Object state transitions facilitate understanding use case behavior because a use case may invoke objects for service. For example, renting a car will invoke a renter object to check whether he/she can rent the car, invoke a car object to check whether the car is available, and invoke a rental record object to record the rental. Accordingly, inspecting the state transitions of the renter, the car, and the rental record facilitates understanding the behavior of the use case “Rent a car”. Since checking use case behavior facilitates verifying use cases, showing object state transitions facilitates verifying prototypes. To conclude, when a prototype is executed, its use cases are selected to execute. When executing a use case, users provide input data and verify the use case by inspecting output data and state transitions of objects invoked by the use case. A prototype is regarded as verified if all its use cases are verified.

3.2 The language

Figure 16 shows a subset of the BNF-like grammar of the language, which depicts that a prototype is primarily composed of use cases and classes (Grammar 1 in Figure 16). Use cases model the use case diagram, whereas classes model the class diagram. As to the activity diagrams, they are modeled as statements in use cases. Grammar 1 also shows that non-primitive activities may also exist. The modeling of a non-primitive activity is not described below because it is similar to a use case. Differences between modeling a use case and a non-primitive activity are described as follows:

1) The keyword used in a use case is “usecase” whereas that in a non-primitive activity is “activity” (Grammar 2).
2) Exceptions can be associated with a use case but cannot be associated with a non-primitive activity.

```
1. Prototype ::= {UseCase | Class | NonPrimitiveActivity}  /* A prototype is composed of use cases, classes, and non-primitive activities */       (a)
2. NonPrimitiveActivity ::= “activity” ActivityName “(“ (Parameter) “)” “{“ {Data} {Statement} “}”
3. UseCase ::= “usecase” UseCaseName “(“ (Parameter) “)” “{“ {Data} {Statement} (Exception) “}”
4. Exception ::=  “exception” ExceptionName “{“ {Statement} “}”
5. Statement ::= ForEachStat |  RetrieveStat | FlagStat | InvokeObjectOp | InvokeNonPrimitiveStat | InvokeUseCase
                 | ClassInstance | ConcurrencyStat | SelectionStat | Relationship | GeneralStat
/* The “InvokeObjectOp” statement invokes object operations. */
/* The “InvokeNonPrimitiveStat” invokes non-primitive statements */
/* The “InvokeUseCase” statement invokes use cases. It can be used to implement the “use” and “extend” relationships among use cases */
/* “GeneralStat” are general statements such as “if” statement, assignment statements, and so on */
/* “Relationship” define relationships among objects */
/* “ClassInstance” instantiates objects from classes */
6. ForEachStat ::= “for each” ObjectName “in” ClassName “{“ {Statement} “}”
7. RetrieveState ::= “retrieve” ObjectName “from” ClassName” “with” AttCond “;”
/* “AttCond” limits the object to retrieve, e.g,, “retrieve . . . with name == “aa” retrieves the object whose attribute “name” has the value “aa” */
8. ConcurrencyStat ::= “concurrency” “{“ { “{“ {Statement} “}” } “}”
9.  SelectionStat ::= “selection” “{“ { “{“ {Statement} “}” } “}”
10. FlagStat ::= SetFlag | ReceiveFlag | SignalFlag | WaitForFlag
11. SetFlag ::= “setflag” FlagName “(“ (Parameter) “)” “;”
12. ReceiveFlag ::= “flagentry  “(“ (Parameter) “)” “:”
13. SignalFlag ::= “signalflag “(“ (Parameter) “)” “;”
14. WaitForFlag ::= “waitflag” “(“ (Parameter) “)” “;”
15. Class ::= ClassDef   “{“   (Attribute)   Constructor   (Operation)   “}”
     /* A class is composed of  attributes, a constructor, and operations. */
16. ClassDef ::= “class”   ClassName   [“extends”   ClassName]
    /* “extends” defines inheritance relationships*/
17. Operation ::= [DataType]   OperationName   “(“   (Parameter)   “)”   “{“   {Statement}   “}”
```

(b)

Symbol	Meaning
::=	is defined as
|	alternative
[X]	zero or one instance of X
(X)	zero or more instance of X
{X}	one or more instance of X
/* . . . */	comments
un-quoted symbols	non-terminals
quoted symbols	terminals

Figure 16. A subset of BNF-like syntax for a prototype. (a) Syntax, (b) Symbols used in the syntax

The modeling of use cases and classes in a prototype are respectively described below.

3.2.1 Use cases

A use case is composed of data definitions, executable statements, and zero or more exceptions (Grammar 3). Parameters can be passed in/out a use case. Example 1 shows the use case “rentACar”, which models the use case “Rent a car” in Figure 11. Statements used in the use case will be described later. Moreover, a detailed explanation of the use case can be found in section 4. The example shows that the use case consumes three input parameters and reacts to the exception “unRent”. From Figure 11 and Example 1, one can see that the mapping between an activity diagram and the corresponding language description is clear.

Example 1. The use case “rentACar”

```
usecase rentACar(in String renterID, in String carID, in String mortgageItem) {
   renterClass renter;
   carClass car;
   mortgageRecordClass mortgageRecord;
   String msg1, msg2;

   retrieve renter from renterClass with ID == renterID;
   retrieve car from carClass with ID == carID;
   if(renter.member == ”y”) { // the renter is a member
      concurrency {
      {
         {
            setflag checkMemF(renter);
            waitflag checkMemF(msg1);
         }
         {
            setflag checkCarF(car);
            waitflag checkCarF(msg2);
         }
      } // end of concurrency
      if (msg1 == “ok”) and (msg2 == “available”) {
         rentTheCar(renterID, carID); // invoke the non-primitive activity “rentTheCar”
      }
   }
   else { // the renter is not a member
      setflag checkCarF(car);
      waitflag checkCarF(msg2);
      if (msg2 == “available”) {
         selection {
            { // selection 1
               setflag addMemF(renter);
               rentTheCar(renterID, carID);
            }
            { // selection 2
               mortgageRecord = new mortgageRecordClass(renterID, mortgageItem);
               rentTheCar(renterID, carID);
            }
         }
      }
   }

   exception unRent {
      // If the exception “unRent” occurs, invoke the use case “rentalRecovery” to recover the rental.
      rentalRecovery(renterID, carID);
   }
}
```

As shown in Example 1, activities of a use case are described using executable statements. In addition to general statements such as the “if” statement, our language provides the following statements for use cases:

1) The “for each” statement (Grammar 6) is used when every instance of a class should be managed in the same way. This statement can be used to model multiple triggers in an activity diagram. For example, the multiple trigger in Figure 9 can be modeled using the “for each” statement in Example 2. The example depicts that every car in the “carClass” will be manipulated as follows: (1) check the car’s age and (2) create a status record for the car.

Example 2. A use case containing the “for each” statement

```
usecase checkCarStatus(){ // check the status of every car
   // data declarations
   carClass car;
   rentalRecordClass rentalRecord;
   carStatusClass carStatus;

   for each car in carClass {
      flagentry checkCarF(in carClass car):
      // On receiving the flag “checkCarF”, the use case is executed from here.
      if car.year>10 {
         // Set the flag “removeCarF” to remove a car that is older than ten years.
         // The flag will be received by the use case “removeACar”.
         setflag removeCarF(car);
      }
      // If the use case is initiated by the flag “checkCarF”, the use case finishes here.
      signalflag checkCarF(car.status);

      // Add a status record for each car
      retrieve rentalRecord from rentalRecordClass with carID = car.ID;
      if (rentalRecord != -1) { // the car is rented
         carStatus = new carStatusClass(car.ID, “rented”, rentalRecord.renterID,
                 rentalRecord.deadline);
      }
      else { // the car is not rented
         carStatus = new carStatusClass(car.ID, “available”, NULL, NULL);
      }
   } // end of for each statement
} // end of use case
```

2) The “retrieve” statement (Grammar 7) retrieves an instance of a class that matches a condition. The condition generally limits the values of one or more attributes of the class. For example, the following statement retrieves a renterClass instance whose ID value equals to “renterID”. The retrieved instance is set to the variable “renter”. If the retrieval fails, the value “-1” will be placed into “renter”.

```
retrieve renter from renterClass with ID == renterID;
```

3) Flag statements model use case communication. The “setflag” statement (Grammar 11) sets flags. Setting a flag causes a use case or parts of a use case to execute. Parameters can be associated with the “setflag” statement. For example, the statement “setflag checkCarF(car);” in Example 1 sets the flag “checkCarF” and passes the parameter “car”.

The “flagentry” statement (Grammar 12) receives flags. It actually is an entry of a use case. For example, the flag “checkCarF” set by the use case “rentACar” (see Example 1) will be received by the use case “checkCarStatus” (see Example 2) by the statement “flagentry checkCarF(in carClass car):”. When the use case “checkCarStatus” receives the flag “checkCarF”, the use case will be executed starting from the “flagentry” statement. Note that a use case has one and only one normal entry, which is for normally executing the use case. The use case, on the other hand, can have arbitrary number of flag entries for receiving flags, in which a flag entry is for a flag. In this regard, setting a flag corresponds to executing the statements starting from the corresponding flag entry.

The “signalflag” statement (Grammar 13) signals a flag, which means that the activities required to execute by a flag is finished. For example, the statement “signalflag checkCarF(car.status);” in Example 2 depicts that if the use case “checkCarStatus” is executed by receiving the flag “checkCarF”, the execution ends in this statement. That is, the flag “checkCarF” cause the following statements to execute (see Example 2).

```
flagentry checkCarF(in carClass car):
if car.year>10 {
   // Set the flag “removeCarF” to remove a car that is older than ten years.
   setflag removeCarF(car);
}
signalflag checkCarF(car.status);
```

Note that the “flagentry” and “signalflag” statements will be executed only if the use case is executed by receiving flags. If multiple flag entries are defined in a use case, only the corresponding “flagentry” and “signalflag” statements will be executed when a flag is received. A “signalflag” statement can return values if necessary. For example, the statement “signalflag checkCarF(car.status);” returns the value “car.status”.

The “waitflag” statement (Grammar 14) waits for flags. It is used when a use case can proceed only when a flag is signaled. For example, the statement “waitflag checkCarF(msg2);” in Example 1 depicts that after setting the flag “checkCarF”, the use case “rentACar” can proceed only when that flag is signaled. The “waitflag” statement can get return values. For example, the statement “waitflag checkCarF(msg2);” gets the return value “msg2”.

4) The statement to invoke an object operation is used when an activity of a use case should invoke that operation. This statement has the syntax “object_name.operation_name(parameters);”.

5) The statement to invoke a use case is used when the use case should be invoked to accomplish another use case. This statement is generally used when the “use” or “extend” relationships exist among use cases. The statement has the syntax “use_case_name(parameters);”.

6) The statement to instantiate an instance from a class is used when the instance is needed. For example, the following statement (see Example 1) instantiates an instance from the class “mortgageRecordClass”.

```
mortgageRecord = new mortgageRecordClass(renterID, mortgageItem);
```

7) The “concurrency” statement (Grammar 8) is use to specify concurrent execution of statements. For example, the following statements (see Example 1 and Figure 11) depict that the flags “checkMemF” and “checkCarF” are set and waited in parallel.

```
concurrency {
{
   {
      setflag checkMemF(renter);
      waitflag checkMemF(msg1);
   }
   {
      setflag checkCarF(car);
      waitflag checkCarF(msg2);
   }
} // end of concurrency
```

8) The “selection” statement (Grammar 9) is used to specify selections. For example, the following statements (see Example 1) model two selections depicted in Figure 11.

```
selection {
   { // selection 1
      setflag addMemF(renter);
      rentTheCar(renterID, carID);
   }
   { // selection 2
      mortgageRecord = new mortgageRecordClass(renterID, mortgageItem);
      rentTheCar(renterID, carID);
   }
}
```

As mentioned in section 2.2, exceptions may occur during the execution of a use case. The “exception” statement (Grammar 4) defines exceptions and their handlers. Example 1 shows the handler of the exception “unRent” defined in the use case “rentACar”. The handler is composed of a statement to invoke the use case “rentRecovery” as shown in Example 3.

Example 3. The use case "rentalRecovery"

```
usecase rentalRecovery(in String rID, in cID){
   rentalRecordClass rentalRecord;
   mortgageRecordClass mortgageRecord;

   retrieve rentalRecord from rentalRecordClass with
              renterID == rID and carID == cID;
   retrieve mortgageRecord from mortgageRecordClass with
              renterID == rID;
   concurrency
      if (rentalRecord != -1) rentalRecord.remove();
      if (mortgageRecord != -1) mortgageRecord.remove();
   }
}
```

3.2.2 Classes

As shown in Grammar 15 of Figure 16, a class is composed of attributes, constructor, and operations. The constructor of a class, which has the same name as the class, is an operation to instantiate instances from the class. Note that each class has the implicit operation "remove" to remove an instance of the class. Example 4 models the class "renterClass", which depicts that it has two attributes, namely "ID" and "member". Moreover, the class has an operation "changeToMember" other than the constructor.

Example 4. The class "renterClass"

```
class renterClass {
   String ID, member;

   renterClass(in String renterID, in String memberOrNot){
      ID = renterID;
      member = memberOrNot;
   }

   // operations
   changeToMember(){
      member = "y";
   }
}
```

4. EXAMPLE

The use case diagram of the simplified car rental system is shown in Figure 4, in which twelve use cases, two actors, four flags, and one exception are depicted. The activity diagrams of some use cases are respectively shown in Figures 6 through 13. Moreover, the class diagram is shown in Figure 15. See the previous two sections for the explanation of those diagrams.

A prototype based on those diagrams is shown in Appendix 1. To prevent wasting space, the appendix shows only partial use cases and classes. The use case “rentACar” is traced below to facilitate understanding the prototype. Please take Figure 11 as a reference in the following tracing.

In the beginning of the use case, the information of the renter and the car is retrieved from the corresponding classes. The use case then checks whether the renter is a member. If he/she is a member, the use case concurrently sets and waits for the flags “checkMemF” and “checkCarF” (see the concurrency block). The former flag causes the use case “checkRentalRecord” to check whether a bad record is associated with the member. The latter flag causes the use case “checkCarStatus” to check whether the car is available. After the flags are signaled (i.e., after the “waitflag” statements are finished), the use case “rentACar” inspects the checking result, if no bad record is associated with the renter and the car is available, the renter rents the car. The renting is accomplished by invoking the non-primitive activity “rentTheCar”.

If the renter is not a member, the use case “checkACar” first checks whether the car is available by setting and waiting for the flag “checkCarF”. If the car is available, a selection block is used for the renter to decide whether he/she wants to become a member. If he/she want to become a member, the use case sets the flag “addMemF” to inform the use case “ addAMember” to register the renter as a member. The renter then rents the car. If the renter does not want to be a member, he/she should mortgage something valuable before he/she rents the car.

In addition to the above normal activity, the use case “rentACar” reacts to the exception “unRent”. When the exception occurs, the use case “rentalRecovery” will be executed to handle the exception.

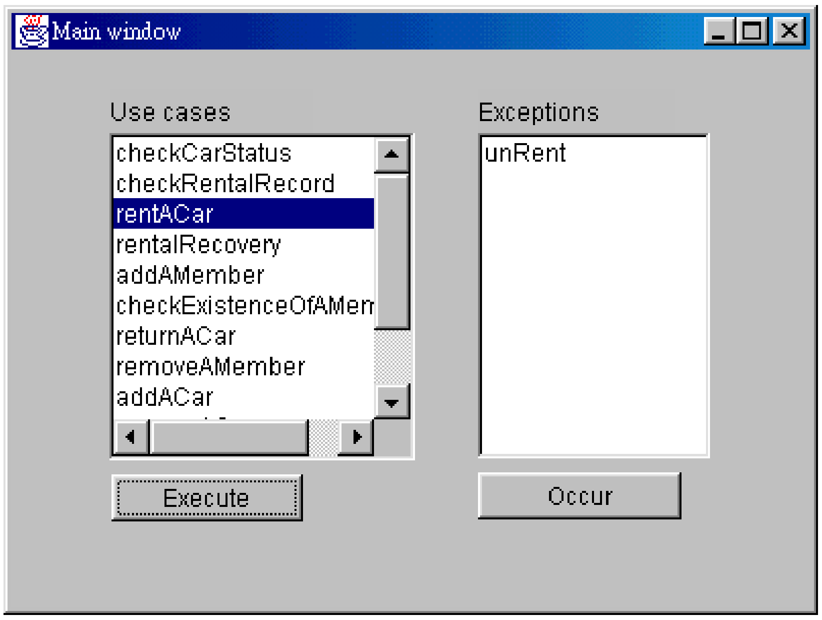
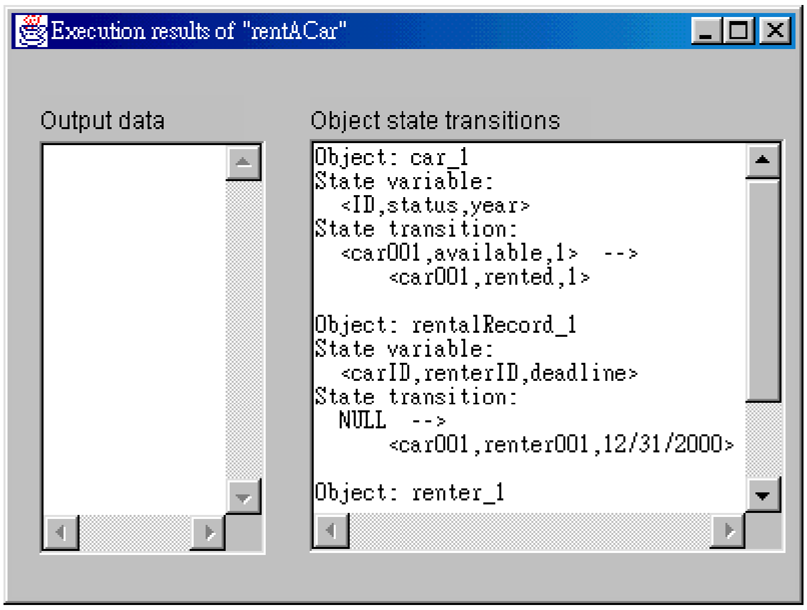


Figure 17. The main window to execute a prototype



```
Object: car_1
State variable:
  <ID,status,year>
State transition:
  <car001,available,1>  -->
      <car001,rented,1>

Object: rentalRecord_1
State variable:
  <carID,renterID,deadline>
State transition:
  NULL  -->
      <car001,renter001,12/31/2000>

Object: renter_1
```

Figure 18. Execution results of the use case “rentACar”

Figure 17 shows the window for executing a prototype, the left side lists the use cases to execute, and the right side lists the exceptions that may occur during prototype execution. Figure 18 shows the window after the execution of a use case. The left side displays output data, while the right side depicts object state transitions. In displaying an object state transition, variables used in the transition are first displayed. The variables are generally the attributes of the object. The state transition is then displayed, in which the state before the symbol "-->" is the initial state whereas that after the symbol is the state after the use case has been executed. Note that the special state "NULL" denotes an un-existing object. With this, the state transition of the object "rentalRecord_1" in Figure 18 means that the object is created after the use case is executed.

5. COLCLUSIONS

This article proposes a technique to verify requirements before they are modeled into a specification. With this technique, requirement workers, including the customers, analysts, end users, domain experts, and so on, first identify requirements according to their perspectives. The captured requirements are then analyzed and integrated in a meeting. The meeting results (i.e., the requirements after integration) are then represented in our requirement model, which is composed of use case diagram, activity diagram, and class diagram. Requirements represented in the model can be transformed into a prototype using our prototyping language. The prototype can then be executed to verify the requirements. During the verification, use cases are selected to execute, When executing a use case, users (e.g., the analysts or customers) provide input data. They then inspect output data and object state transitions (which facilitate understanding use case behavior) to verify the use case. When all use cases are verified, the prototype is verified.

Our technique offers the following features:

1) Requirement errors can be identified before a specification is produced. Since requirement errors will propagate to the corresponding specification, correcting requirement errors reduces possible specification errors.
2) The requirement model models necessary components of requirements, including use cases and their relationships, use case communication, exceptions, and domain objects and their relationships.
3) The mapping between the requirement model and the prototyping language is clear.

Therefore, using the language to prototype requirements should be easy.

ACKNOWLEDGMENT

This research is sponsored by the National Science Council in Taiwan under grant number NSC89-2213-E-259-012

REFERENCES

1. Luqi and W. Royce, "Status Report: Computer-Aided Prototyping", *IEEE Software*, pp. 77-81, November, 1991.
2. S. -C. Chou and J. -Y. Chen, "An Object-Oriented Analysis method Based on parallel Decomposition of Function and Data", *Report on Object Analysis and Design*, vol 2, No 6, pp22-35, 1996.
3. S. -C. Chou and J.-Y. J. Chen, "An Object-oriented Analysis Technique Based on Unified Modeling Language", to appear in the *Journal of Object-Oriented Programming*.
4. P. G. Wijayarathna, Y. Kawata, A. Santosa, K. Isogai, M. Maekawa, "GSL: A Requirements Specification Language for End-user Intelligibility", *Software - Practice and Experience*, vol. 28, no. 13, pp. 1387-1414, 1998.
5. M. B. Ozcan, "Use of Executable Formal Specifications in User Validation", *Software - Practice and Experience*, vol. 28, no. 13, pp. 1359-1385, 1998.
6. S. -C. Chou, J. -Y. Chen, and C. -G. Chung, "An Executable Specification Language for Specification Understanding in Object-Oriented Specification Reuse", *Info. Software Technolo.*, Vol 38, No 6, pp419-434, June 1996.
7. S. Tyszberowicz and A. Yehudai, "OBSERV - A Prototyping Language and Environment", *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 3, pp. 269-309, july 1992.
8. M. Baldassari and G. Bruno, "PROTOB: An Object Oriented Methodology For Developing Discrete Event Dynamic Systems", *Computer Languages*, vol. 16, no. 1, pp. 39-63, 1991.
9. R. -J. Lea and C. -G. Chung, "Rapid Prototyping from Structured Analysis: Executable Specification Approach", *Information and Software Technology,* vol. 32, no. 9, pp. 589-597, 1990.
10. Luqi, V. Berzins, and R. Yeh, "A prototyping language for real-time software",

IEEE Transactions on Software Engineering, vol. 14, no. 10 , pp. 1409-1423, October 1988.

11. I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
12. Martin Fowler and Kendall Scott, UML Distilled, Applying the Standard Object Modeling Language, Addison-Wesley, 1997.

APPENDIX 1. A prototype of the simplified car rental system.

```
usecase checkCarStatus(){ // check the status of every car
   // data declarations
   carClass car;
   rentalRecordClass rentalRecord;
   carStatusClass carStatus;

   for each car in carClass {
      flagentry checkCarF(in carClass car):
      // On receiving the flag “checkCarF”, the use case is executed from here.
      if car.year>10 {
         // Set the flag “removeCarF” to a car that is older than ten years.
         // The flag will be received by the use case “Remove a car”.
         setflag removeCarF(car);
      }
      // If the use case is initiated by the flag “checkCarF”, the use case finishes here.
      signalflag checkCarF(car.status);

      // Add a status record for each car
      retrieve rentalRecord from rentalRecordClass with carID = car.ID;
      if (rentalRecord != -1) { // the car is rented
         carStatus = new carStatusClass(car.ID, “rented”, rentalRecord.renterID,
                 rentalRecord.deadline);
      }
      else { // the car is not rented
         carStatus = new carStatusClass(car.ID, “available”, NULL, NULL);
      }
   } // end of for each statement
} // end of use case

usecase checkRentalRecord() {
   renterClass renter;
   rentalRecordClass rentalRecord;
   badRecordClass badRecord;
   for each renter in renterClass with renter.member == “y” {
      flagentry checkMemF(in renterClass renter):
      for each rentalRecord in rentalRecordClass with renterID = renter.ID {
         if (rentalRecord.deadline < system.currentDate) {
            signalflag checkMemF(“bad”);
            badRecord = new badRecordClass(renter.ID, rentalRecord.carID,
                         rentalRecord.deadline);
         }
      }
      signalflag checkMemF(“ok”);
   }
}

usecase rentACar(in String renterID, in String carID, in String mortgageItem) {
   renterClass renter;
   carClass car;
   mortgageRecordClass mortgageRecord;
   String msg1, msg2;

   retrieve renter from renterClass with ID == renterID;
   retrieve car from carClass with ID == carID;
   if(renter.member == ”y”) { // the renter is a member
```

```
        concurrency {
        {
            {
                setflag checkMemF(renter);
                waitflag checkMemF(msg1);
            }
            {
                setflag checkCarF(car);
                waitflag checkCarF(msg2);
            }
        } // end of concurrency
        if (msg1 == “ok”) and (msg2 == “available”) {
            rentTheCar(renterID, carID); // invoke the non-primitive activity “rentTheCar”
        }
    }
    else { // the renter is not a member
        setflag checkCarF(car);
        waitflag checkCarF(msg2);
        if (msg2 == “available”) {
            selection {
                { // selection 1
                    setflag addMemF(renter);
                    rentTheCar(renterID, carID);
                }
                { // selection 2
                    mortgageRecord = new mortgageRecordClass(renterID, mortgageItem);
                    rentTheCar(renterID, carID);
                }
            }
        }
    }

    exception unRent {
        // If the exception “unRent” occurs, invoke the use case “rentalRecovery” to
        // recover the rental.
        rentalRecovery(renterID, carID);
    }
}

usecase rentalRecovery(in String rID, in cID){
    rentalRecordClass rentalRecord;
    mortgageRecordClass mortgageRecord;

    retrieve rentalRecord from rentalRecordClass with
                renterID == rID and carID == cID;
    retrieve mortgageRecord from mortgageRecordClass with
                renterID == rID;
    concurrency
        if (rentalRecord != -1) rentalRecord.remove();
        if (mortgageRecord != -1) mortgageRecord.remove();
    }
}

activity rentTheCar(in String renterID, in String carID) {
    rentalRecordClass rentalRecord;
    renterClass renter;
    carClass car;
    rentalRecord = new rentalRecordClass(renterID, carID);
    retrieve renter from renterClass with ID == renterID;
    retrieve car from carClass with ID == carID;
    // establish relationships
    renter associateWith rentalRecord;
    car associateWith rentalRecord;
}

usecase addAMember(renterID) {
    String msg;
    renterClass renter;

    // invoke the use case "checkExiatenceOfAMember"
    checkExiatenceOfAMember(renterID, renter, msg);
    if (msg == "existing but not member") {
        falgentry addMemF(renter);
        renter.changeToMember();
        signalflag addMemF();
    }
    else if (msg == "not existing") {
        renter = new renterClass(ID, "y"); // create a renter and set it to be a member
    }
}

usecase checkExistenceOfAMember(in String renterID, out renterClass renter, out
         String msg) {
    retrieve renter from renterClass with ID == ID;
    if (renter == -1) {
        msg = "not existing";
    }
    else if (renter.member == "y") {
        msg = "existing";
    }
    else {
        msg = "existing but not member";
    }
    return;
}

// other use cases are omitted

class carClass {
    // define attributes below
    String ID, status;
    int year;

    // constructor
    carClass(in String carID, in String carStatus, in int carYear) {
        ID = carID;
        status = carStatus;
        year = carYear;
    }

class renterClass {
    String ID, member;

    renterClass(in String renterID, in String memberOrNot){
        ID = renterID;
        member = memberOrNot;
    }

    // operations
    changeToMember(){
        member = “y”;
    }
}

// other classes are omitted
```