

Getting from use cases to code Part 1: Use-Case Analysis

Gary Evans
Independent Object Technology Evangelist, Evanetics
13 Jul 2004

from The Rational Edge: The first in a two-part series, this article presents a case study that analyzes the requirements captured in use cases and transforms them into implementable representations that can be directly coded.

Writing use cases to capture the software requirements that are visible to system actors has been a common practice since Ivar Jacobson introduced them in 1992. But a common confusion has accompanied this practice. Once I have my use cases, how do I use them to get to my code? This two-part article series will present a case study that analyzes the requirements captured in use cases and transforms them into implementable representations that can be directly coded. My goal is to sufficiently clarify this transformation so that you can immediately apply these ideas to your current, or next, software project.

The IBM Rational Unified Process® (RUP®) advocates writing use cases to capture the operational requirements of a software system.[1] Use cases are actually a constituent of a larger requirements package of documents collectively known as the Software Requirements Specification (SRS), which contains all the requirements for a software project. The SRS includes the following requirements artifacts:

- *Use Case Model, which consists of:*
  1. *Use case diagram: A visual representation of system users (actors) and the services they request from the system.*
  2. *Actor definitions: A textual description of the requestors of services provided by your system, and services provided to your system.*
  3. *Use case descriptions: Textual descriptions of the major services provided by the system under discussion.*
- *Supplementary Specification: A document that captures the system-wide requirements, and those functional aspects of the system which are neither visible to the system's actors, nor local to a specific use case.*

These requirements artifacts become inputs to the subsequent analysis and design activities of the Analysis and Design discipline in RUP. Exactly which requirements artifacts are produced, of course, depends on the forces driving your development effort. If you are doing "hot fixes" (i.e., critical bug fixes on a product already in production) you might not have any requirements documents, only bug reports that indicate the released software does not meet its originally stated requirements. If you are doing a maintenance or enhancement release of software (i.e., adding new functionality to an existing product) you might have one or two use cases describing how these new functions interact with a user, but you would not have a Supplementary Specification because no changes to the non-functional properties of the software have occurred.

In this discussion I am assuming a brand-new, "green-field" development project for software that does not yet exist. This will be an object-oriented project using the Unified Modeling Language (UML) to represent concepts and relationships. I am also assuming that the reader is comfortable with the concepts of class and object, and is at least comfortable with reading UML version 1.x or 2.0 class diagrams, sequence diagrams, and collaboration diagrams.

Use-Case Analysis activity

This narrative will focus on the Use Case Analysis activity in RUP. As you can see in Figure 1, this activity incorporates artifacts which are normally produced in the RUP Architectural Analysis activity.

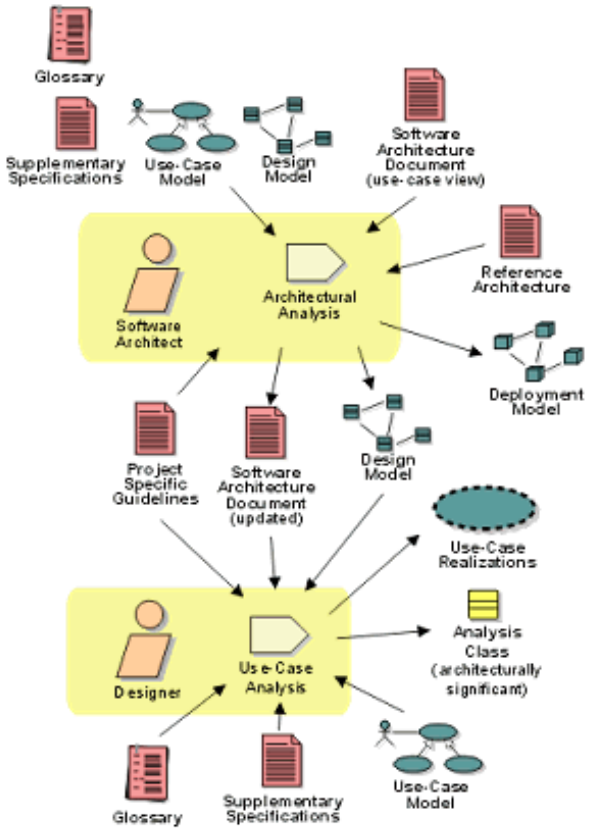


Figure 1: Workflow for Architectural Analysis (early Elaboration)

Admittedly, a rigorous approach to describing the software development process would embrace the larger architectural perspective of enterprise systems, and questions of reuse. But I will not be formally discussing the Architectural Analysis activity for three reasons:

1. My goal is to focus on the lower-level artifacts and activities used and carried out by developers, not architects.
2. It would expand the scope of this series beyond a reasonable length for a non-book publication, and
3. It is my experience as a process and architecture consultant that the discipline of performing architectural analysis is not embraced by a large percentage of software development organizations. If you are currently doing architectural analysis, then you will have already performed some of the steps I cover in this article. It is always commendable to take an architectural perspective on a new, or large, project. But if you are not currently embracing architectural analysis, then the approach in this series will illustrate the minimal steps that should help you gather some of this needed information.

The purpose of the Use-Case Analysis activity is:

- To identify the classes that perform the various flows of events in a use case.
- To distribute the use-case behavior to those classes, using use-case realizations.
- To identify the responsibilities, attributes, and associations of the classes.
- To note the usage of architectural mechanisms to provide the functionality needed by the use case, and the software system in general.

We can alternately say that the goal of Use-Case Analysis is to take our understanding of the requirements in the system's use cases and iteratively transform those requirements into representations that support the business concepts, and meet the business goals of those requirements. In Use Case Design we will transform these business concepts into classes, objects and relationships, components, interfaces, etc., which can be implemented in an executable environment.

The diagram in Figure 2 is taken from the RUP Analysis and Design Activity Overview, which illustrates where the Use Case Analysis activity occurs within the context of the other Analysis and Design activities.

Figure 2: Use-Case Analysis activity in RUP

Use Case Analysis is composed of several steps in RUP [RUP2003]:

- For each use case in an iteration
    1. Create a use case realization
    2. Supplement the Use-Case descriptions (if necessary)
    3. Find Analysis Classes from Use-Case Behavior
    4. Distribute Behavior to Analysis Classes
- For each resulting analysis class
    1. Describe the Class's Responsibilities
    2. Describe the Class's Attributes and Associations
        - Define Class Attributes
        - Establish Associations between Analysis Classes
        - Describe Event Dependencies between Analysis Classes
- Reconcile the Use Case Realizations
- Establish Traceability
- Qualify Analysis Mechanisms
- Evaluate the Results of Use-Case Analysis

Please note that the order of these steps is not cast in stone. The actual sequence you follow may differ according to your understanding of the domain you are analyzing, your experience with RUP or UML, your personal preferences for the models you use, or the metaphor you follow for characterizing the properties of your analysis classes (e.g., responsibility-centric, behavior-centric, or data-centric approaches). What is important is that you achieve a comprehensive expression of the *problem* you are to solve (note that we achieve a comprehensive definition of the *solution* we have chosen in Use-Case Design, which is the subject of Part 2 in this series). I will follow most, but not all, of these steps in this article, and I will change the sequence somewhat. As I discuss each step, I will explain why I have found a slightly different sequence to be beneficial when teaching object-oriented analysis and design (OOAD) to people who are new to RUP and OOAD.

As Figure 3 illustrates, there are some specific activities that separate the writing of a use case from its implementation in code. This illustration also shows the steps recommended by RUP within the context of Use Case Analysis. This diagram will become our visual roadmap as the remainder of this paper addresses the specific tasks within these activities.

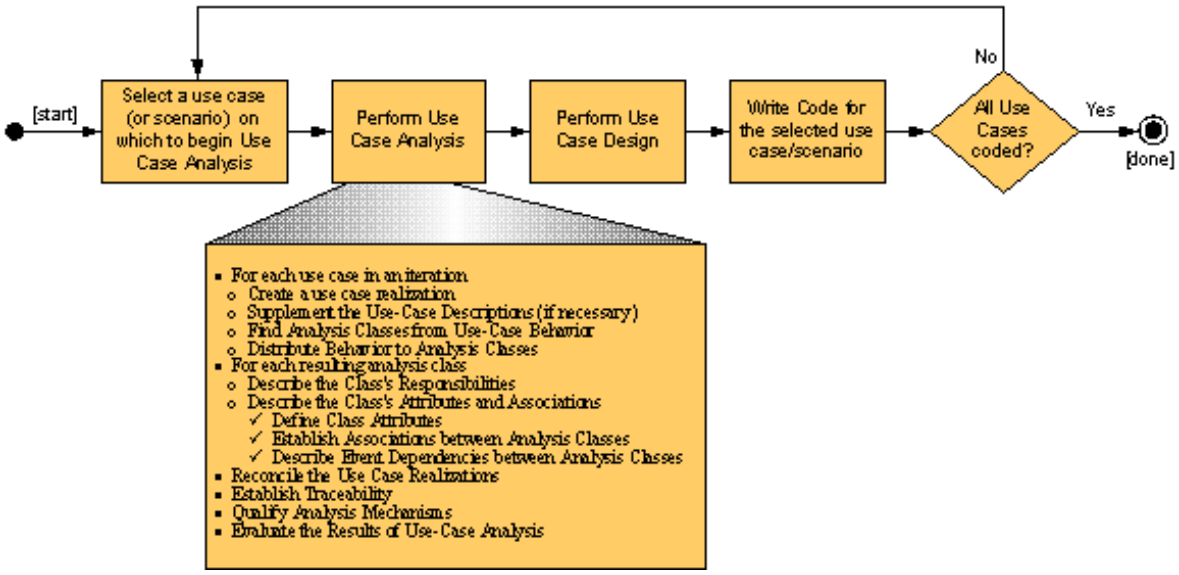


Figure 3: The steps of use case analysis

Example use case

To facilitate our understanding of what we do with use cases after we discover and develop them, we need to start with a sample use case. Consider this very brief use case for a hypothetical browser-based software system for an auto rental company. Such a system may have a half-dozen or so use cases allowing their customers to request various services, such as:

- Reserve a Vehicle
- Cancel a Reservation
- View Rental History
- View or Edit Customer Profile
- Join Awards Program, etc.

To simplify our modeling, I will assume that our rental agency does not support corporate accounts, only individual accounts.

To keep our example simple and understandable, we will focus on just one of these use cases. Here is a use case description for the use case: *Reserve a Vehicle*.

Use Case: Reserve a Vehicle.

1. This use case begins when a customer indicates he wishes to make a reservation for a rental car.
2. The system prompts the customer for the pickup and return locations of the reservation, as well as the pickup and return dates and times. The customer indicates the desired locations and dates.
3. The system prompts for the type of vehicle the customer desires. The customer indicates the vehicle type.
4. The system presents all matching vehicles available at the pickup location for the selected date and time. If the customer requests detail information on a particular vehicle, the system presents this information to the customer.
5. If the customer selects a vehicle for rental, the system prompts for information identifying the customer (full name, telephone number, email address for confirmation, etc.). The customer provides the required information.
6. The system presents information on protection products (such as damage waiver, personal accident insurance) and asks the customer to accept or decline each product. The customer indicates his choices.
7. If the customer indicates “accept reservation,” the system informs the customer that the reservation has been completed, and presents the customer a reservation confirmation.
8. This use case ends when the reservation confirmation has been presented to the customer.

This use-case description is necessarily generic: it is not specific to a Web-based application, nor is it specific to the situation where a human being walks up to a rental counter and requests a vehicle to rent. This description addresses only the *what*, not the *how*, of the system — what is the behavior of the system and the corresponding behavior of the use-case actor (i.e., the customer). If you substitute “customer service representative” for “the system” above, you will have a

reasonably accurate description of what happens when a human walks into the rental office to get a vehicle. In this case, the reservation confirmation presented to the customer in Step 7 is the printed rental agreement.

Alternately, if you are planning on implementing a Web-based interface, this use case describes that approach also, if you recognize that multiple steps in a use case can be combined into a single browser page (e.g., steps 2 & 3 would most certainly be on the same page). In the Web environment, the reservation confirmation presented to the customer in step 7 is the confirmation number associated with the rental transaction, presented to the actor on the transaction summary Web page.

Also note the style of the use case. It is written in active voice and present tense. Active voice is clear and emphatic, while passive voice is a weaker presentation. E.g., “John throws the ball” is active voice. The doer of the action, John, precedes the verb. The passive voice equivalent of this sentence is: “The ball is thrown by John,” or just “The ball is thrown,” leaving the thrower unspecified. Here the doer of the action, John, follows the verb. Invariably, in passive voice, the doer is contained within a prepositional phrase initiated with the word “by.” Keep your use case descriptions clear and consistent. Use active voice, present tense. Use a limited and clear vocabulary. Do not introduce unnecessary words, and be consistent. For example, don’t use the word “customer,” then “client,” then “business patron” just to be creative. Your reader will conclude that you might be discussing three separate actors, with different security profiles and authorizations!

Now that we have this use case as a starting point, let’s follow the RUP steps of Use Case Analysis.

Use-Case Analysis step 1: Create a use-case realization

The first step in RUP’s Use-Case Analysis is to create what RUP calls a *use case realization*. Before we get into a formal definition of a realization, let’s step back and ask, “What really is a use case? and What do we need to validate our use case?” Our written use case is a description of a process: a business process for allowing a customer to reserve a vehicle from our business. It states that we will follow a certain flow of events (step B occurs after step A, etc.), and we will enforce certain business rules, such as not processing a rental request unless we get a first name and last name of the renter, and not processing a rental request for a vehicle which is not available at the pickup location on the specified date.

Since we are doing an object-oriented software system, the behavior of our use case must be carried out by the classes and objects in our system. But so far we don’t have any classes or objects yet, so we have to discover the classes that we will need to carry out the process in our use cases. And we have to specify which classes will interact to provide the behavior we have designated in our use case.

As Figure 4 illustrates, a use-case realization is really a collection of several UML diagrams which together validate that we have the classes, responsibilities, and object interactions necessary to provide the behavior in our use case process.

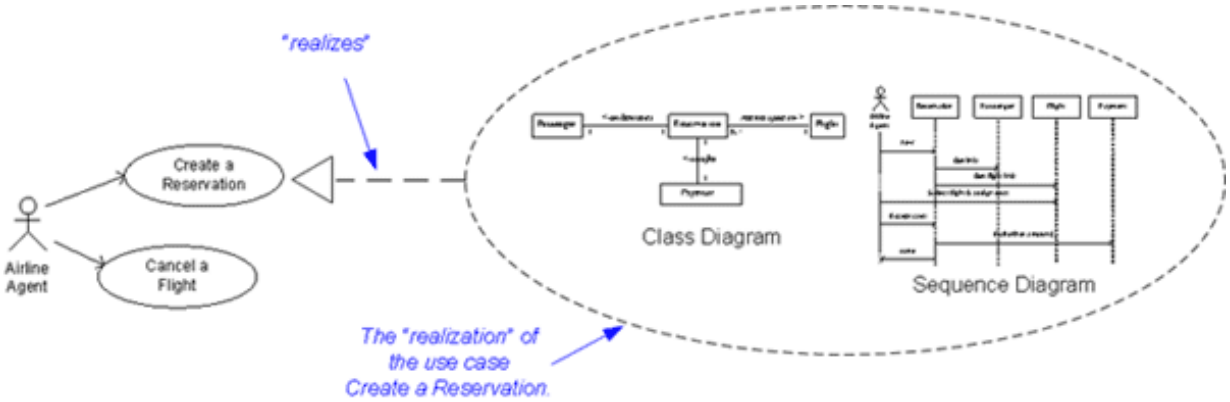


Figure 4: A RUP use-case realization for an airline reservation system

Specifically, a use-case realization is normally represented using:

- A UML *class diagram* for those classes participating in the use case on which we are focusing (sometimes called a *View of Participating Classes* class diagram.), and
- One or more UML *interaction diagrams* to describe which objects interact, and the interfaces which these objects will invoke to carry out the work of the use-case process. UML defines two types of interaction diagrams: a *sequence diagram* (shown in Figure 4), and a *collaboration diagram*. Either diagram can be effective.

This sounds like a lot to do in this first step, doesn’t it? Yes, and actually this first step is a housekeeping task when you are using a CASE tool such as Rational Rose or Rational XDE, and really means “create a place to hold your use-case realization.” We will develop the actual class and interaction diagrams later in this process. But now we know the content we will be developing to populate our use-case realization: a class diagram and one or more interaction diagrams.

Use-Case Analysis step 2: Supplement the use-case descriptions

While you are in an analysis mindset, your use case description will generally address only the behavior of the system that is externally visible to an actor using the system. It is quite acceptable to describe in summary fashion some of the internal, non-visible behavior of the system, but do not attempt to design your

system in the use-case description.

As an example, consider Step 4 in our use case: “The system presents all matching vehicles available at the pickup location for the selected date and time. If the customer requests detail information on a particular vehicle, the system presents this information to the customer.” Hmmm, do we have a data source that will provide these matching vehicles? We might know that the vehicle schedules are maintained for all rental locations by a CICS application on an MVS mainframe accessed by LU6.2 APPC, but let’s not be so specific yet. Let’s make it clear that *what* we have to do is go outside the boundary of our reservation system, without specifying how we expect to do this. Here is the same Step 4, now *supplemented* to indicate a new data source we will simply call *Vehicle Inventory*: “…the system accesses the vehicle inventory for the pickup location, and presents a new page with all vehicles of the specified categories available at the pickup location for the selected date and time.”

Here we have specified that there is an external data source of vehicle information, and made high-level reference to presentation via Web pages. This was an isolated example of supplementing, but our use-case reader can now get a better understanding of the total geography of behavior involved in the use case.

In an iterative development process such as RUP you move from analysis to design in a very short time. In a 4-week iteration in the Construction phase (mid-project), you may spend the first week capturing your requirements, doing your Analysis and Design disciplines, and then spend the last three weeks writing and testing the code for the iteration. Your analysis-level use-case descriptions will focus on *what* behavior the system will exhibit, but you might need to enhance those descriptions to include more internal interactions so your customer or business analysts can be confident you have not left out significant business processing. Bear in mind that you want to supplement the use-case descriptions only to the point that you can effectively determine the analysis classes in your system. The identification of design-level classes (e.g., trees, stacks, queues, collections, etc.) should be deferred to a later discipline (i.e. *design*).

Example: Supplement the Reserve a Vehicle use case

Let’s assume our system will be a browser-based Web application. We want to provide our customers with on-line capability to make their own vehicle rental reservations whenever it is convenient for them to do so. We may need to supplement our use-case description to make it more specific to our target environment, without going overboard into design (that comes later).

Here is the *Reserve a Vehicle* use case in a more specific expression, still focusing on *what* is done, rather than *how*:

Use Case: Reserve a vehicle to a customer (supplemented)

1. This use case begins when a customer enters our rental Website page.
2. The system presents fields to prompt the customer for the pickup and return locations of the reservation, and the pickup and return dates and times. The customer indicates his desired locations and dates. The system also presents an option for the customer to limit the vehicle search to specific categories of vehicles — e.g., compact, SUV, full-size, etc. The customer may specify a search category, or categories, if desired. The default is to search for all categories of vehicles. If the customer is participating in our rental awards program, he may enter his awards identification number in a separate field on the page. If this field is filled in, the system will access the customer’s rental profile, which the system will retain to pre-populate any required information.
3. If the customer indicates he wishes to continue the reservation session, the system accesses the vehicle inventory for the pickup location, and presents a new page with all vehicles of the specified categories available at the pickup location for the selected date and time. With each vehicle the system presents a base rate, which may be discounted based on the customer’s rental profile. If the customer requests detail information on a particular vehicle, the system obtains this information from the vehicle inventory and presents it to the customer.
4. If the customer selects a vehicle for rental, the system presents a new page which prompts for information identifying the customer (full name, telephone number, email address for confirmation, credit card issuer, etc.). If a customer profile exists, the system pre-populates all known values. Some fields are mandatory; others (such as email address) are optional. The customer provides any remaining required information. The system also presents information on protection products (damage waiver, personal accident insurance, etc.) and their daily costs, and asks the customer to accept or decline each product. The customer indicates his choices.
5. If the customer indicates “accept reservation,” the system presents a page summarizing the reservation (type of vehicle, dates and times, any elected protection products and their charges, total rental cost), and presents the customer a reservation confirmation. If the system has an email address for the customer, the system sends a reservation confirmation to that address.
6. This use case ends when the reservation confirmation has been presented to the customer.

In this supplemented version we are clearly describing the behavior of a browser-based application, specifying a fair amount of behavior that is not visible to the customer actor. But there is no design-level information in the use case yet.

Is it necessary to provide this additional detail for every use case?

No, it’s not. But remember that “detail” means detail — not implementation. The goal is to get just enough detail to understand the analysis classes you will need in your system, and to get consensus from your customer or business analysts that your use case meets their goals. If your first cut at a use case description is a bit thin in helping you identify some analysis classes, then do the supplemented use-case.

Caution: It’s not easy to find this middle ground between abstract specification at one extreme, and implementation specification at the other. It takes time and practice. Work with it, find help, and remember it’s better to err toward abstraction if you are not sure how detailed you should be. It’s easier to add some detail that you missed than to rummage around in a quagmire of implementation details from which you will find it almost impossible to extricate yourself.

Why should I do the high-level use case at all? Why not just do a supplemented use case?

The answer is, the abstract use case (light on internal behavior) is the most generic description of behavior. What if you wanted to do a client/server version of the Reserve a Vehicle use case? If you started with a browser-specific version, you would have to re-write the whole thing from scratch when you changed your target platform. The generic version is technology-agnostic, and that is a great value when you are not ready, or able, to specify the production environment. Additionally, the abstract version lets your Business Analysts or Subject Matter Experts focus on what the system’s business behavior will be, rather than the implementation which they may not understand at all.

Use-Case Analysis step 3: Find analysis classes from use-case behavior

According to RUP, the purpose of this step is to identify a candidate set of analysis classes which will be capable of performing the behavior described in our use cases. So far we don't have any classes, so our main goal will be to identify the analysis classes we need in our Auto Rental system.

But this raises a very interesting and important question: Just what is an *analysis class*? There are two answers, really. First, a *business-level analysis class* is one that is essential to the business domain, without reference to implementation or technology constraints. For example, a banking system has Bank Customer, Account, Account Transaction, etc., and it does not matter if this is a new e-commerce system or a savings and loan system from the 1890s.

Second, RUP extends this definition by defining analysis classes in three disjoint categories: as *entity*, *controller*, and *boundary* classes. RUP's entity classes are roughly equivalent to the business-level analysis classes above. Controller classes are process-aware, and sequence-aware: they control and direct the flow of control of an execution sequence. It is common to find a controller class enforcing the process behavior of a use case. Boundary classes mediate the transfer of information and events between the software being executed and the outside world. Boundary classes handle the input and output functions required by a software system.

In my experience teaching object technology and modeling, I have found that teams employing RUP's entity, controller, and boundary categories jump too quickly into a design mindset, without performing adequate analysis of the problem they are trying to solve. In fact, it is quite clear that controllers and boundary classes are actually technology classes, not business classes. They are part of the solution domain defined in design, not part of the problem domain described in analysis. So, in this article I am going to concentrate on the business-level, technology-agnostic analysis classes, and leave alone almost all technology issues until we discuss design. Be aware that the activity of finding these business-level classes is normally performed in RUP's Architectural Analysis activity — if your project is pursuing that degree of RUP conformance.

With that said, let's recall that the focus of a use case description is *behavior* — what services the system will provide to the actors who are requestors of those services. There is nothing object-oriented about use-case descriptions, but these descriptions can be used to discover the classes or objects in our system. Classes can be discovered in many different ways, from different sources:

- General domain knowledge
- Previous systems that are similar
- Enterprise models / Reference architectures
- CRC (Class/Responsibility/Collaborator) sessions
- Glossary of terms
- Data mining

One simple technique for discovering classes is known as *grammatical dissection*, and I will illustrate that. In grammatical dissection we identify the nouns in our requirements. Of these nouns (and adjective-noun pairs):

- Some will become classes.
- Some will become attributes of a class.
- Some will have no significance at all for our requirements.

Let's identify and underline the nouns (skipping pronouns such as "he") in our supplemented use case for *Reserve a Vehicle*, as follows:

Use Case: Reserve a Vehicle to a customer (Supplemented).

1. This use case begins when a customer enters our rental Website page.
2. The system presents fields to prompt the customer for the pickup and return locations of the reservation, and the pickup and return dates and times. The customer indicates his desired locations and dates. The system also presents an option for the customer to limit the vehicle search to specific categories of vehicles — e.g., compact, SUV, full-size, …. The customer may specify a search category, or categories, if desired. The default is to search for all categories of vehicles. If the customer is participating in our rental awards program, he may enter his awards identification number in a separate field on the page. If this field is filled in, the system will access the customer's rental profile, which the system will retain to pre-populate any required information.
3. If the customer indicates he wishes to continue the reservation session, the system accesses the vehicle inventory for the pickup location, and presents a new page with all vehicles of the specified categories available at the pickup location for the selected date and time. With each vehicle the system presents a base rate, which may be discounted based on the customer's rental profile. If the customer requests detail information on a particular vehicle, the system obtains this information from the vehicle inventory and presents it to the customer.

4. If the customer selects a vehicle for rental, the system presents a new page which prompts for information identifying the customer (full name, telephone number, email address for confirmation, credit card issuer,….). If a customer profile exists, the system pre-populates all known values. Some fields are mandatory; others (such as email address) are optional. The customer provides any remaining required information. The system also presents information on protection products (damage waiver, personal accident insurance, etc.) and their daily costs, and asks the customer to accept or decline each product. The customer indicates his choices.

5. If the customer indicates “accept reservation,” the system presents a page summarizing the reservation (type of vehicle, dates and times, any elected protection products and their charges, total rental cost), and presents the customer a reservation confirmation. If the system has an email address for the customer, the system sends a reservation confirmation to that address.

6. This use case ends when the reservation confirmation has been presented to the customer.

Note that every occurrence of any noun, or adjective-noun pair, has been underlined. We have lots of duplicates, so gather the distinct nouns/pairs into a single list in Table 1, sorted alphabetically:

Table 1: Candidate nouns/entities

1	address	21	page
2	awards identification number	22	pickup location
3	awards program	23	product
4	base rate	24	protection products
5	categories	25	rental cost
6	charges	26	rental profile
7	choices	27	reservation
8	credit card issuer	28	reservation confirmation
9	customer	29	reservation session
10	customer profile	30	return location
11	daily costs	31	search category
12	dates	32	system
13	detail information	33	telephone number
14	email address	34	times
15	field	35	use case
16	information	36	values
17	locations	37	vehicle inventory
18	name	38	vehicle search
19	new page	39	vehicle
20	option	40	Website page

How do we identify which of these candidate nouns really describe classes in our problem domain? A very usable approach is to challenge each candidate noun with a few simple questions shown in Figure 5:

1. Is this candidate inside our system boundary?
   If not, it might be an actor of our system.
2. Does this candidate have identifiable behavior for our problem domain?
   (i.e., can we name the services/functions that are needed in our problem domain and that this candidate would own and provide?)
3. Does this candidate have identifiable structure?
   (i.e., can we identify some set of data this candidate should own and manage?)
4. Does this candidate have relationships with any other candidates?

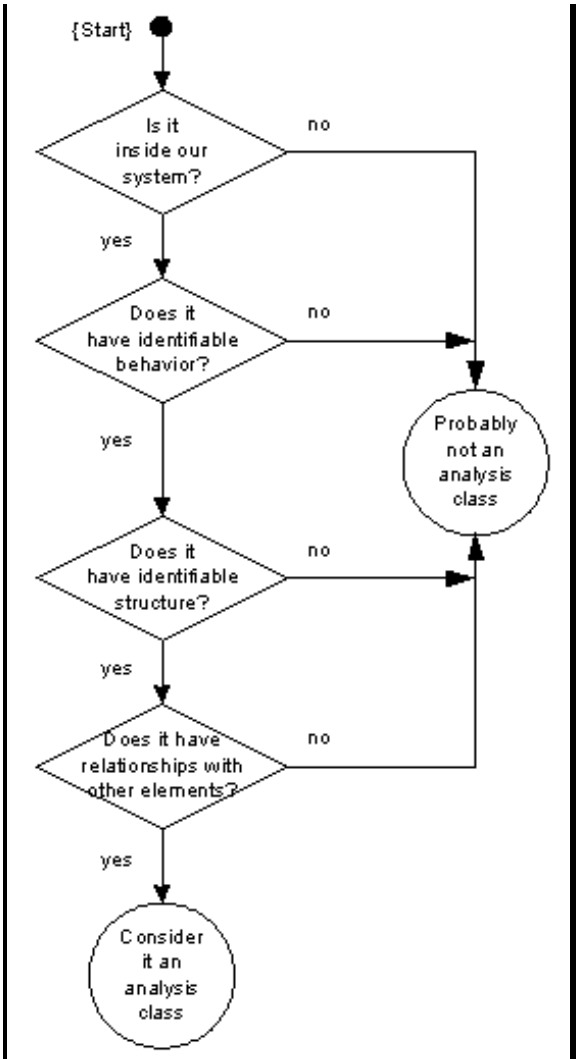


Figure 5: Questions for discovering analysis classes

If you find a “no,” then the candidate is probably not a class; move on to the next candidate. If the answer is “yes,” keep asking the questions. If you get all “yes” answers, conclude the candidate is a class, and get the next candidate to evaluate.

If we challenge each of our candidates with these questions, we should obtain results similar to Table 2:

Table 2: Noun challenge results

	Candidate Noun	Business Glossary Meaning	Is a class?	Justification
1	address	A location of residence for a rental branch, or a customer.	No	Attribute of another class such as Customer or Rental Branch. [Declaring this to be not a class is a simplifying assumption for the model I am building.]
2	awards identification number	A unique identifier that may be associated with a customer.	No	No behavior; simple attribute of a customer, used to obtain pre-filed customer profile information.
3	**awards program**	Program to accrue credits for future rentals, etc.	Yes	Manages policy rules for rental awards.
4	base rate	Nominal cost of a vehicle rental; may be discounted by customer criteria such	No	Simple attribute/value.

		by customer criteria such as AAA, AARP, etc.		
5	categories	Organizing criterion for types of vehicles: e.g., economy, compact, full-size, pickup, etc.	No	Simple attribute/value.
6	charges	Itemized cost for a reservation or protection product.	No	Simple attribute/value in Reservation or Protection Product.
7	choices	Denotes multiplicity in protection products offered.	No	Choice of protection products is captured in the relationship between Reservation and Protection Products.
8	credit card issuer	Financial provider of credit card payments to rental company: VISA, Discover, etc.	No	Out of scope. Our system only knows how to interface to Electronic Funds Transfer network.
9	**customer**	The human responsible for the rental.	Yes	The Customer object (not the human customer) manages information about human responsible for the rental transaction.
10	**customer profile**	Collection of information, preferences associated with the renter.	Yes	Manages information about values for specific attributes of a rental relationship.
11	daily costs	Charges applied for each day the vehicle is in possession of renter.	No	Simple attribute/value.
12	dates	Calendar date for rental and return.	No	Simple attribute/value.
13	detail information	Specific properties of a vehicle	No	Attributes of vehicle.
14	email address	Email address of renter.	No	Simple attribute/value.
15	field	Area on a computer screen.	No	Not a problem domain class; artifact of a Web page.
16	information	Unknown. Too vague.	No	Vague; non-specific.
17	locations	A brief identifier for a Rental Branch, such as "Airport," or "downtown."	No	'Location' really represents an attribute, an abbreviation, of a rental branch. Could be confused with 'address.' See **rental location.**
18	name	Name of customer doing the renting	No	Attribute/value of Customer which is association with a Reservation
19	new page	A computer screen.	No	In analysis we will focus on the business domain classes. Pages will be addressed in distributing behavior to analysis classes.
20	option	Opportunity to specify vehicle search criteria.	No	Simple attribute/value used for vehicle search.
21	page	A computer screen.	No	In analysis we will focus on the business domain classes. Pages will be addressed in distributing behavior to analysis classes.
22	pickup location	Rental branch that	No	Synonymous with 'rental location.'

				behavior to analysis classes.
22	pickup location	Rental branch that removes a rented vehicle from its inventory.	No	Synonymous with 'rental location.'
23	product	A protection product that a renter may elect to include in a reservation for liability protection.	No	Synonymous with 'Protection Product.' Use that full name because our business people prefer that.
24	**protection product**	A protection product that a renter may elect to include in a reservation for liability protection.	Yes	(protection) products provide services to enforce business rules, know how to price themselves, and maintain data on whether they have been accepted, declined, their legal requirements by state, etc. They have relationships to rented vehicles and reservations.
25	**rental location (added)**	A business location at which vehicle rentals and returns occur.	Yes	Already part of business vocabulary. More accurate term than just 'location.'
26	rental cost	Total cost of a reservation.	No	Attribute/value of Reservation
27	rental profile	Set of rental properties and values associated with a Customer.	No	Synonymous with 'Customer Profile'
28	**reservation**	An agreement between a Rental Branch and a Customer. Becomes binding when the Customer receives possession of the vehicle.	Yes	Many relationships to other candidates; has behavior to manage associated vehicle, protection products; knows its total cost, its associated customer....
29	reservation confirmation	A pledge by the rental company that the reserved vehicle will be available for the Customer.	No	Attribute/value representing the state of a Reservation.
30	reservation session		No	Not an analysis class. (Will later represent a design concept.)
31	return location	Rental branch that adds a rented vehicle to its inventory.	No	Synonymous with 'rental location.'
32	search category	Organizing criterion for types of vehicles: e.g.,	No	Simple attribute/value. Same as vehicle category.
33	system	The software system we are building.	No	System is the thing we are describing.
34	telephone number	A telephone number.	No	Simple attribute/value.
35	times	Hour/minute property of a vehicle pickup or return.	No	Simple attribute/value.
36	use case	The description we are creating.	No	Use case is not part of the system; it is not in the system itself.
37	values	Content of customer rental profile.	No	Simple attribute/value.
38	**Vehicle Inventory**	Fleet of vehicles at a Rental Branch.	Yes	Represents the set of vehicles assigned to a Rental Branch; manages its list of vehicles, knows which vehicles are available, in

				manages its list of vehicles, knows which vehicles are available, in repair, etc.
39	vehicle search	Properties to use in a search of vehicles (e.g., "compact automobile" only)	No	Criteria (values) used to search for vehicles in the Vehicle Depot, and the behavior of doing the search.
40	**vehicle**	Any asset which can be rented.	Yes	Represents a revenue-generating, physical vehicle. Has behavior to rent itself, express its status, etc.
41	Website page	Brower-based presentation of information.	No	Not an analysis class. (Will later represent a design concept.)

Note that *rental location* has been added although it was not part of the use case. In talking with our Subject Matter Experts (SMEs), we learned that the normal business vocabulary uses 'location' to refer to both an address, and to a rental branch. To resolve the ambiguity, we agreed to use the term *rental location* for the business location where rentals and returns are conducted.

From this list we extract those candidates that we have designated "yes." This yields the following list of *analysis classes*:

award program	rental location
customer	reservation
customer profile	vehicle
protection product	vehicle inventory

Wow! That's only eight analysis classes versus the thirty nine candidates we started with. The four questions have helped us rapidly narrow our focus — and that's a good thing.

But what if we made a mistake? What if we missed a "real" class, or we included a class that we should not have? It doesn't matter, really. The iterative nature of RUP will reveal our errors, and allow us to correct them with minimum damage to work we have already done. The goal of analysis and design is not to "get it all right up front." The goal is to get it right when you need to have it right. Getting started is often the hardest part of any task, and we have now made the leap from having no objects to having objects (or from having no classes to having classes). What is important is that we have started, and we can begin to move forward in an object-oriented perspective.

We now have completed the first three steps in RUP's Use Case Analysis activity:

- For each use case in an iteration
    1. Create a use case realization
    2. Supplement the Use-Case Descriptions (if necessary)
    3. Find Analysis Classes from Use-Case Behavior

If we follow RUP rigorously, the next RUP step will be:

    4. Distribute Behavior to Analysis Classes

Again, I am going to deviate a bit from the standard RUP flow, with the following justification: Consider where we are. We have just identified eight entities that we believe are classes in our Auto Rental system. Before we do anything else, we need to add content to these eight entities to be sure they *are* classes.

There are three basic approaches for "fleshing out" our analysis classes:

- A data-driven approach
- A behavior-driven approach, or
- A responsibility-driven approach.

The *data-driven* approach is very popular with people coming from a database, or procedural, background. They see the world in terms of data, and data relationships, and tend to populate their classes first with data — usually with no strategy on how to assign operations (i.e., functions) to the class. This is fine, but data is only half of the total picture. Indeed, the very concept of a class involves the intimate association of data with the operations that manipulate that data.

The *behavior-driven* approach takes this dual nature into account. It populates a class first with the operations that the class will perform, and determines from those operations the data that should be owned by the class. Very good, but how do I make sure that the operations I assign to a class are coherent? And how do I distinguish between operations and classes, so that I know *this* operation belongs in *this* class, but *that* other operation should be in a different class? We need a filter, some kind of discriminator that will help us make good decisions about our operations. This filter is what the responsibility-driven approach gives us.

A *responsibility-driven* approach starts with a large-grained view of the class and first assigns responsibilities to that class. This approach describes first the "mission" statement of a class in the context of the problem domain in which we are working. This mission statement is a declaration of the services the class will provide to requestors of those services. In military terms, the responsibilities are strategic; the operations and data are tactical (subservient to, and a means of achieving, the strategy).[2]

And once we identify our class responsibilities, we should construct an analysis class diagram to capture the structure of the relationships among our classes. This structure is usually inherent in the business domain we are modeling, and a UML analysis class diagram gives us a visual representation of this relationship structure.

So, the deviation I am recommending produces the following change to the standard RUP flow (sequence changes are in **bold**):

- For each resulting analysis class
    1. Describe the Class's Responsibilities
    2. **Establish Associations between Analysis Classes (analysis class diagram)**
    3. **Distribute Behavior to Analysis Classes (discover operations)**
    4. Describe each Class's Attributes and Associations
        - Define Class Attributes
        - Describe Event Dependencies between Analysis Classes

Use-Case Analysis step 4: Describe the class's responsibilities

This step is done for each analysis class we have identified. A responsibility of a class describes the services that this class will provide in our system, and that no other class will provide. Responsibilities in different classes must not overlap.

Based on our understanding of our vehicle rental domain, and in consultation with our vehicle rental SMEs and business analysts, we can document responsibilities for each analysis class, as shown in Table 3.

Table 3: Responsibilities for each analysis class

Class name	Description of the Class	Class Responsibilities
Award Program	Represents a set of business rules which offer 'points' based on completed and paid vehicle rentals. The accrued 'points' can be assigned monetary value as surrogate currency for future rental agreements.	Manages the value of the award program 'account' for an associated customer. Knows the business rules to apply to a completed rental for accruing 'points.' Knows how to transform 'points' to the appropriate surrogate monetary value for a rental agreement.
Customer	Represents the human individual (no company accounts) who may request to reserve a vehicle.	Manages the information associated with a specific human customer (e.g., email address, physical address, telephone, etc.).
Customer Profile	Represents a set of properties describing the rental preferences for the associated Customer.	Manages its attributes and values as a cohesive set of properties associated with a given Customer. Knows the Customer for which it manages these properties.
Protection Product	Represents a legally binding liability- or risk-management agreement between the rental company and the customer. The legal agreement does not become binding until the rental reservation is confirmed, and the customer takes possession of the vehicle.	Each protection product knows its rules for application based on the geographic state (e.g., Georgia, Nevada, etc.) in which the rental is made. Knows the allowed limits on liability and property coverage.
Rental Location	Represents a business facility where vehicles are rented and returned. A rental location has an assigned vehicle inventory.	Knows its physical address. Has a unique identifier within the rental company. Manages its associated vehicle inventory to determine what automobiles are available for a prospective reservation.
Reservation	Represents a non-binding agreement to 'reserve' a vehicle that has been requested by a customer.	Each reservation uniquely identifies the association between a Customer and a Vehicle for a specific date and time.
Vehicle	Represents a physical vehicle that is part of a rental location vehicle inventory.	Knows its status (rented, damaged, ...). Knows the vehicle inventory it is part of; or the reservation it has been assigned to. Knows its schedule for availability.
Vehicle Inventory	Represents the vehicle fleet assigned to a rental location for given dates and times.	Manages the vehicles in the inventory for a specific Rental Location (i.e., the vehicles physically present, and transient vehicles that are scheduled for return, but have not been returned yet).

James Rumbaugh et al.[3] defines an object, or class, as “a concept, abstraction, or thing with *crisp boundaries* and meaning for the problem at hand [my *emphasis*].” It is primarily through the definition of responsibilities that you can give a class “crisp boundaries,” a clear definition of what it does, and does not, do.

What if we made a mistake in our responsibilities? Again, it doesn't matter. We have a starting point and we will move forward. It is common to adjust the responsibilities of classes as we learn more about our system. This is just another example of where we use refactoring to help us build better models and better software.

Use-Case Analysis step 5: Establish associations between analysis classes

Now that we have defined our class responsibilities, we will develop an initial UML class diagram to identify the relationships among our analysis classes. There are four simple tasks we must conduct to develop a class diagram:

1. Identify the classes to be modeled (we have already done this).
2. Identify which of these analysis classes have some kind of relationship with each other.
3. For any two classes that have some relationship, identify the *semantics* of the relationship: is it association, aggregation, composition, or inheritance?
4. For non-inheritance relationships, identify the multiplicity on the relationship. (Multiplicity is an indication of "how many objects of that class might be related on one object of this class?" It's very similar to cardinality in the data modeling world.)

By applying these steps, in conjunction with our identified class responsibilities, we arrive at the UML class diagram shown in Figure 6.

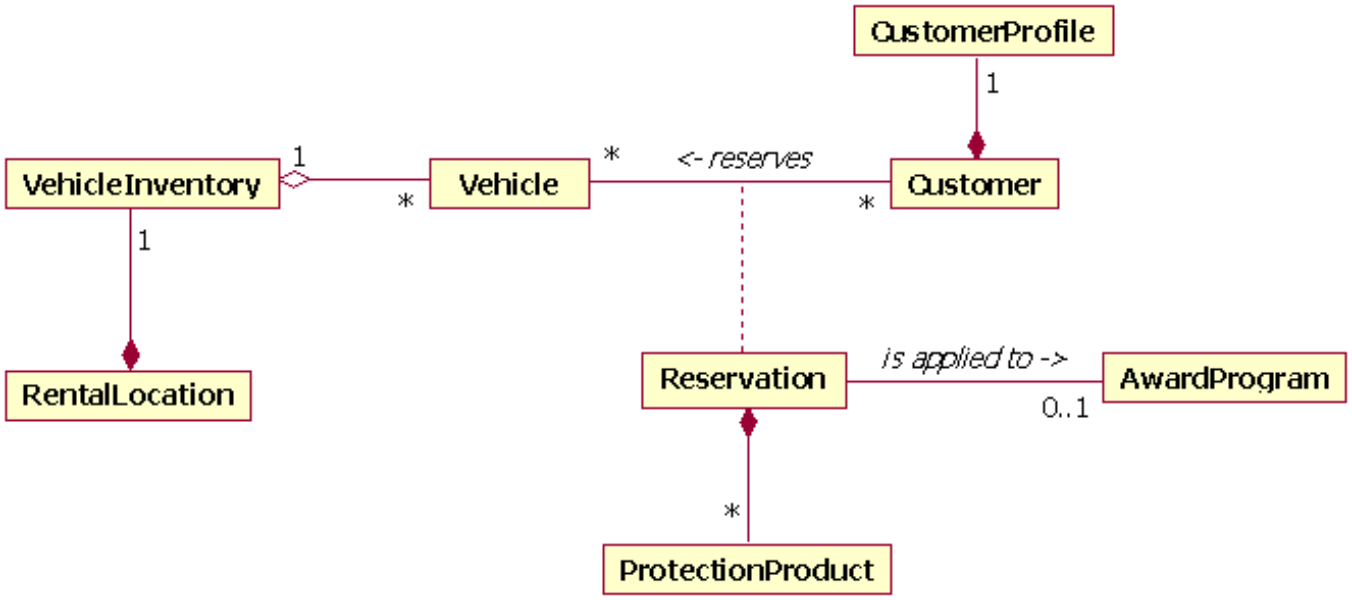


Figure 6: Analysis class diagram for vehicle rental system

There are three types of UML relationship shown on this class diagram, indicated by different line styles. The simple solid line indicates an *association* relationship. This is used to indicate that the two connected classes are in a peer-to-peer relationship and that each class can request the services provided by the other class through its operations.

The filled-in diamond on the line between Reservation and ProtectionProduct is called *composition* (or, *non-shareable aggregation*). This relationship is a "whole/part" or "ownership" relationship. In this class diagram the composition symbol means that the Reservation owns and manages the zero-or-more (*) ProtectionProducts that are included in the Reservation. Further, composition dictates that if the Reservation is destroyed, the ProtectionProducts owned by the Reservation must also be destroyed since they have no business significance if they are not part of a Reservation.

The unfilled diamond on the line between VehicleInventory and Vehicle is called *aggregation* (or, *shareable aggregation*). This is also a "whole/part" or "ownership" relationship, but in aggregation we do not destroy the parts (Vehicle) when we destroy the whole (VehicleInventory). This makes sense: just because a particular RentalLocation will no longer rent cars (it will become a service-only location) the Vehicles may be temporarily "orphaned" but we don't destroy the objects representing the vehicles, we just reassign them to another VehicleInventory.

The numbers and '*' symbols at the ends of the relationship lines are called *multiplicity* specifiers. These symbols denotes the number of, for instance, Vehicles associated with one Customer. Or, conversely, the number of Customers associated with one Vehicle. In this class diagram we have multiplicity that says, "for each Customer, we have zero or more (*) Vehicles reserved (either at one time, or over time)." Reading this in the opposite direction, we have "for each Vehicle, it is reserved by no Customer, or possibly many Customers (over time, obviously)."

In analysis we are trying to make sure we can express and understand the problem for which we are building a solution. The analysis class diagram is a vehicle

for Business Analysts and Subject Matter Experts to review with the technical people, and migrate the model toward a proper description of the problem being solved.

Now we have classes, responsibilities, and a class diagram to show the structure of relationships among the classes. But so far we have no “internals” of the classes — no operations and no attributes. And the class diagram is a static picture. How can we be sure that these classes really can carry out the process described in our use cases? Ah, that is what the next step is for, and it is an all-important step because it maps our use-case descriptions into potential operations on our analysis classes.

Use-Case Analysis step 6: Distribute behavior to analysis classes

How will these classes behave and interact to carry out the work of the *Reserve a Vehicle* use case? We show this by creating a UML interaction diagram to capture the interactions between objects of our analysis classes. Recall that UML Sequence diagrams and Collaboration diagrams are each types of interaction diagrams, and are part of our *use-case realization*. In Figure 7, I show an analysis-level Sequence diagram for the *Reserve a Vehicle* use case.

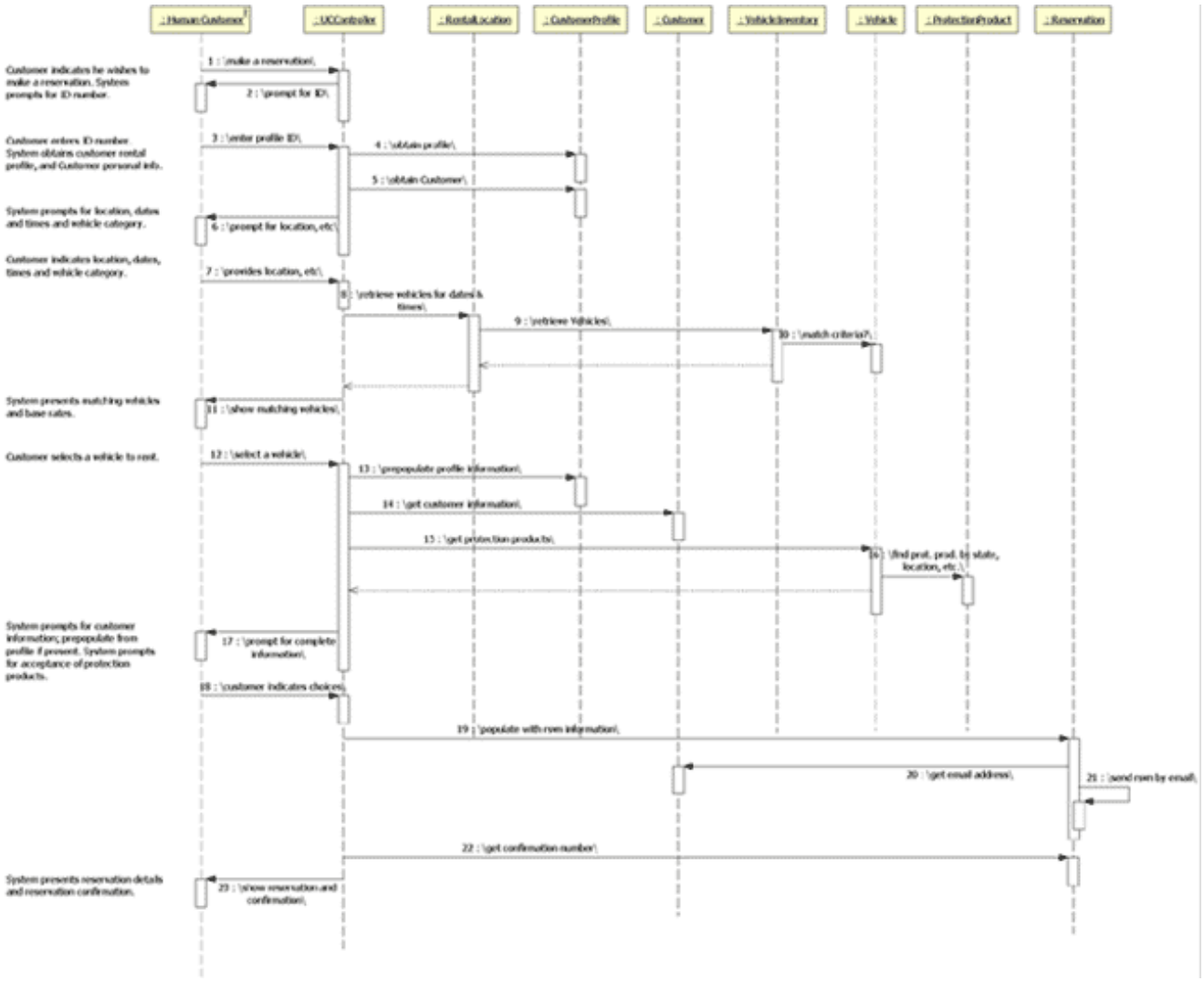


Figure 7: Reserve a Vehicle analysis sequence diagram
Click to enlarge

You will notice that I have introduced a non-business class — UCController — in this diagram. This Use Case Controller class represents a generic placeholder to receive events and messages from the human actor. I have found that most readers of analysis-level Sequence diagrams get very confused when a business class (e.g., RentalLocation or Reservation) acts as a receiver of actor messages. So, I often add to my analysis interaction diagrams a generic Use Case Controller to represent this intelligence, and simplify the reader’s understanding. In design we will rename this to *ReserveAVehicleController*, but for now I want the name to remain generic so *UCController* can be used on every analysis-level Sequence diagram.

Both sequence diagrams and collaboration diagrams contain almost identical information, they just present it differently. Choosing which diagram to use is often a matter of convenience and personal preference. In the sequence diagram the objects are aligned at the top of the diagram, and they have dashed *lifelines* that extend downward. The horizontal arrows with numbered text are called *messages*. In a Sequence diagram, the sequence of messages is shown positionally: time proceeds down the page, so a message low in the diagram is sent after a message that is above it. The messages start on one object’s lifeline, and always terminate on a lifeline, usually another object’s lifeline, but sometimes on the sending object’s own lifeline (see Figure 7, message #21).

The sequence diagram offers a significant advantage over the collaboration diagram, which is the *script* on the left-hand side of the diagram. This text is taken from the use case, or scenario, that the sequence diagram depicts. The script on this diagram is just a terse rendering of the text in the *Reserve a Vehicle* use case. Placing the script into the diagram makes the context of the messages very clear, and links the messages and objects back to the original use case. It is always the case that a given statement in the use case will map to one or more messages sent between the objects in your system. The Sequence diagram makes this explicit.

I want to emphasize a very important characteristic of analysis-level interaction diagrams: *the messages show intent, not implementation, not even interface*. In the *Reserve a Vehicle* Sequence diagram the messages simply indicate what I want done by the receiving object, not the signature of a function call. This deeper interface detail is addressed in design, but now we only want to be sure our classes have the responsibilities to do the work of the use case.

How did we know to send these particular messages to these classes? By following our responsibility definitions. For example, in Step 8, the RentalLocation is asked to meet its responsibility of determining what vehicles are available. In Step 9, the VehicleInventory is asked to retrieve all vehicles for this location that can match the requested rental dates and times. In Step 10, each Vehicle in inventory is asked if it is available to meet these rental criteria. Notice that all of this knowledge is not in the RentalLocation object. We have distributed the intelligence in our system across all of our analysis classes, so that each class can act on a small set of requests that are within its defined responsibilities.

UML note: Objects — To name or not to name?

On the Sequence diagram the object boxes have no name preceding the ":<classname>." These are called anonymous objects. But it is possible to give objects a name. If we have a class called Account we would show it as:

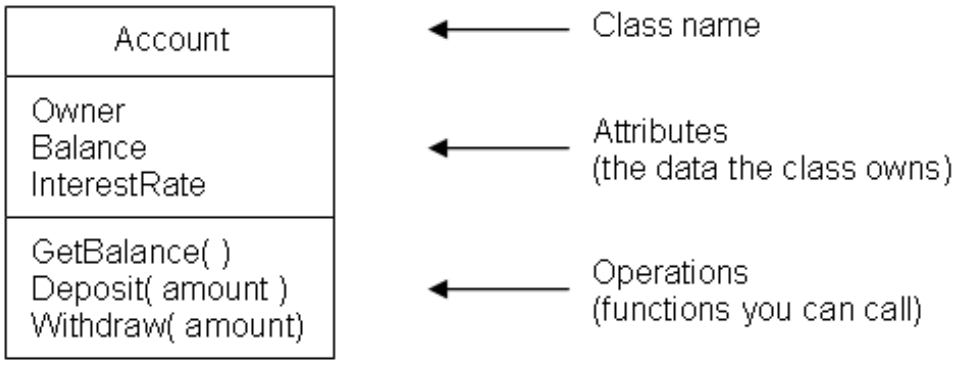
If we create two objects from this Account class definition, *FredsStash*, and *EthelsMadMoney*, they would be shown as:

The one on the left, for example, indicates that "FredsStash is an object of type Account." How do you know if you should name an object or leave it anonymous? If you have a special entity in your system that has a well-known name, you might want to use a named object to represent it. Or, if you want to build a diagram with sample objects (similar to sample tables in a logical data model) you could use named objects. But for most modeling purposes, anonymous objects are sufficient. We are most interested in the services (functions) a class and object provide, and the name of the object does not affect what the class or object can do.

Use case analysis step 7: Describe attributes and associations

In analysis, you will discover some of the attributes (i.e., class data variables) that your classes will need in order to meet their responsibilities. From our list of class responsibilities we can deduce certain attributes for our analysis classes. Additional attributes can be determined from general domain knowledge (e.g., it makes sense that each Vehicle object should have a unique identifier attribute corresponding to the physical vehicle's federally-mandated Vehicle Identification Number).

UML note: Classes in UML have three subcompartments, as shown below, using Account as the class example.



The class diagram shown in Figure 8 shows our vehicle rental analysis classes, the relationships between them, and an initial start on the attributes appropriate to be owned in each class. These are simply the attributes most evident from the class responsibilities. Note that these attributes do not even have datatypes yet because data types are a design issue.

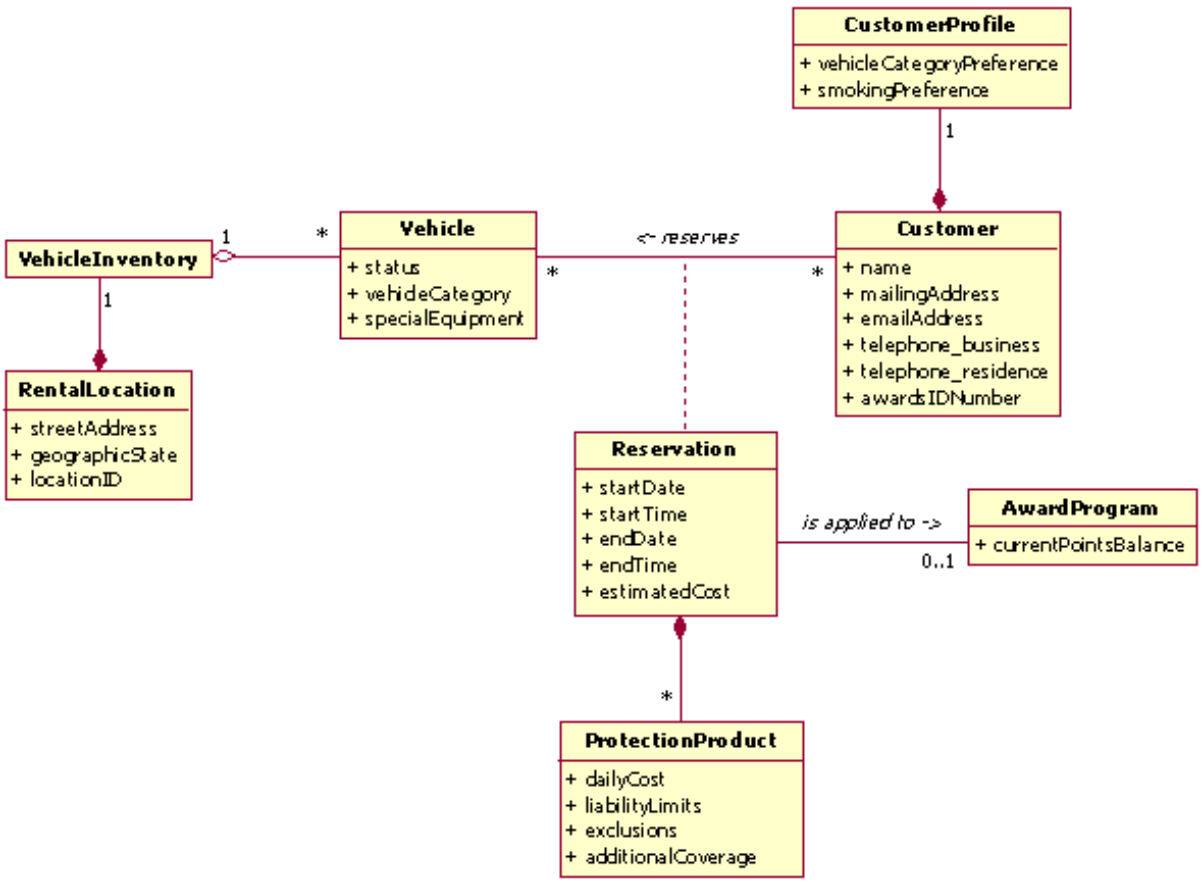


Figure 8: Initial assignment of class attributes

It is enough at this point to only specify that a Customer class has an address attribute. How the address is structured, or even if the address needs to become a class itself, can be decided later. You will notice that VehicleInventory has no attributes yet. This will become an interface to our external, vehicle information repository, and I am not at all sure what data attributes are needed yet. We'll discover them as we move further into the project.

Use-Case Analysis step 8: Qualify analysis mechanisms

An analysis mechanism is a high-level architectural component that provides a service needed by the problem domain, not the technical, solution domain. For example, in an insurance domain it is a business requirement that the information in our Policy, Claim, and other objects must be maintained across usages of a Policy Management application. This business requirement translates into an analysis mechanism called *Persistence:* the maintenance of information and state even when the application is not executing. Note that we do not specify Oracle SQL, or SQL Server, which are specific implementations supporting the function of persistence. We just list persistence, and we will later describe *design mechanisms* and *implementation mechanisms* which will become platform- or vendor-specific.

An example of the relationship among analysis, design, and implementation mechanisms is shown in Table 4:

Table 4: Relationships among analysis, design, and implementation mechanisms

Analysis Mechanisms	Design Mechanisms	Implementation Mechanisms
Persistence	RDBMS	JDBC to SQL Server
Persistence	OODBMS	Ontos, Versant, ObjectStore
Distribution (of objects)	Remote Method Invocation	Java 1.4 from Sun Microsystems.
Distribution (of objects)	Web services	Microsoft .NET
Legacy Interface	CICS application on MVS mainframe	SNA 3270, LU 6.2 APPC

Some common analysis mechanisms are:

- Persistence
- Communication (between processes, or applications)

- Exception handling
- Event notification mechanisms
- Messaging
- Security
- Distribution (i.e., distributed objects)
- Legacy interface

In our Vehicle Rental system, we need to specify analysis mechanisms for:

Persistence	In a normal reservation session, or change reservation session, the Reservation object will undergo change of data and/or state. So we need a persistence mechanism to store the new object data. In analysis we need only to specify a meaningful name for the persistence store, but the details of how we achieve this persistence will be addressed later in design.
Security	We do not want to show any reservation other than that of the correct individual, so we specify a mechanism to assure authentication security.
Legacy Interface	All vehicle information comes from the Corporate Vehicle System, which maintains centralized vehicle information for all rental locations. How we talk with the Corporate Vehicle System depends on the design mechanism choices we can use.

Conclusion

In Part 1 of this series on “Getting from use cases to code,” we have traveled from a single use case, with no knowledge of classes in our system, to a point where we have identified the classes needed to support the goals of that use case, the relationships among those classes, and the attributes needed by those classes, and we have specified several analysis mechanisms that identify services we will need to refine into design and implementation perspectives.

If we repeat this use case analysis process with another use case, we will discover some additional analysis classes, define their responsibilities, new relationships to other analysis classes, perhaps discover new analysis mechanisms, and we will develop another collaboration diagram or sequence diagram to further demonstrate how our classes interact. This demonstrates the incremental aspect of RUP: each task, or iteration, builds on, and adds to, the work done previously.

We have accomplished a lot, but we are not yet ready to begin writing code. Now we are at a point to turn our attention to **Use-Case Design**, which is the subject of Part 2 in this series.

Acknowledgements

I am grateful to Peter Eeles and Zoe Eason of IBM Rational for their insightful comments and suggestions on an earlier draft of this paper.

References

Wirfs-Brock, Rebecca. *Designing Object-Oriented Software*. Prentice-Hall, 1990. A classic in object thinking and modeling. Introduces the significance of the responsibility-driven approach to software modeling and design.

Rumbaugh, Jim, et al. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991, pg. 21. The defining book on the Object Modeling Technique, a major influence on the Unified Modeling Language.

The Rational Unified Process®, version 2003.06.00.65. Rational Software Corporation.

Further Reading

Ambler, Scott. *The Object Primer*, 2nd ed. SIGS, 2001. Covers end-to-end object-oriented development with a single case study.

Fowler, Martin. *UML Distilled*, 3rd ed. Addison-Wesley, 2004. Best introduction to UML (version 2.0) for those learning UML for the first time.

Taylor, David. *Object Technology: A Manager's Guide*. Addison-Wesley, 1998. One of the best introductions to object-thinking ever written.

Bell, Donald. “UML basics — An introduction to the Unified Modeling Language,” in *The Rational Edge*, June 2003: http://www-106.ibm.com/developerworks/rational/library/769.html

Bell, Donald. “UML basics: The activity diagram,” in *The Rational Edge*, September 2003: http://www-106.ibm.com/developerworks/rational/librarycontent/RationalEdge/sep03/f_umlbasics_db.pdf

Bell, Donald. “UML basics: The class diagram,” in *The Rational Edge*, November 2003: http://www-106.ibm.com/developerworks/rational/librarycontent/RationalEdge/nov03/t_modelinguml_db.pdf

Bell, Donald. “UML’s sequence diagram,” in *The Rational Edge*, January 2004: http://www-106.ibm.com/developerworks/rational/library/3101.html

Notes

[1] All references in this series to RUP incorporate the content of RUP version 2003.06.00. All models and code in this series have been generated using IBM Rational Extended Developer Environment (XDE) Developer Plus for Java version 2003.06.

[2] For more information on the responsibility-driven approach, see Rebecca Wirfs-Brock’s book, *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[3] Rumbaugh, Jim, et al., *Object-Oriented Modeling and Design*. Prentice-Hall, 1991, pg. 21.

About the author

Gary K. Evans is the founder of Evanetics, Inc., a consulting company dedicated to reducing risk on software projects through agile techniques and process. He is the author of over a dozen papers on object technology and tools, and is a frequent speaker at major software conferences. He is an avid soccer player, but loves even more working with small development teams, training them in OOAD and agile RUP, and then working with them side-by-side to deliver the right software faster than they ever thought possible.