TWELVE STEPS

TO A BETTER COMPONENT ARCHITECTURE

John Dodd - 1st Draft

ABSTRACT

A systematic process for obtaining an application software architecture is explained, starting from requirements capture. The architecture is defined in terms of software components, which can be built to COM+, CCM, EJB or other distributed component standards.

The process is illustrated using a car rental example, with the diagrams drawn in UML notation.

TWELVE STEPS TO A BETTER COMPONENT ARCHITECTURE

A systematic process for obtaining a component architecture is a vital ingredient of the component-based development approach. An architecture is needed early in the project, so that costs can be estimated, resources can be organized, component purchase options can be investigated and the developers get a good overview of what they are building. A well-formed architecture is needed so that the application remains adaptable, and the components are shareable, reusable, replaceable and enduring.

The process starts with requirements gathering, and the business components are largely derived from these requirements. The process also provides the starting point for user interface design and for component specification and implementation, though these activities will not be covered here.

In figure 1, the steps of the process are classified as structure- or behavior-focused, and have been layered into stages. They are numbered in the order they are generally tackled. In practice, some steps may need to be repeated – due to changing requirements, or because development occurs in increments, or due to errors found down the line.

Stage	Structure	Behavior
Requirements	1.High Level Requirements 2. Concept Model	3. Use Cases 4. Use Case Steps
Analysis	5. Business Type Model	6.Transactions 7. Secondary Use Cases
Architecture	8. Interface Responsibility 9. Interface Dependency 10. Component Architecture Design	11. Operation Interaction 12. Operation Definition
Specification	15. Refined Component Architecture	13. Interface Type Model 14. Pre & Post Conditions

Figure 1: Component Modeling Steps

For brevity, the steps in figure 1 are just named after what they deliver. Generally, a step involves one technique and one deliverable. The organization of project personnel and project phases is outside the scope of this article.

The article will concentrate on the first twelve steps, supported by an example of a Car Rental application. Specification steps 13 to 15 are the subjects of a separate article.

Components and Architectures

Before explaining each step, a few words on what is meant by a component, an architecture and a component architecture.

A *component* is a software building block, used to construct larger components or applications for end-users. A component is a tradable commodity that is separately deployable (though its specification may state it requires other components to function fully). A component offers its functionality through well-defined stable *interfaces*, an interface being a collection of related functions, called *operations* or *methods*. A component has *state* – that is, it remembers information – and this can be made persistent using a database. Client software only accesses the component' s functionality and state (its data) through its interfaces: the internal code, data and database are hidden from all clients. In other words, a component is *encapsulated*. Components are built to run in *container* software, that provides standardized services to the component, such as component activation and transaction management.

An *architecture* defines the big parts of a system, and how they fit together. For component-based applications, the big parts – the architectural units – are predominately software components, although legacy systems and other software packages can be included in the architecture. The presentation layer is another important architectural unit, though it is not usually a formal component. But it often embeds user interface components (sometimes called UI controls). Take a look at figure 2.

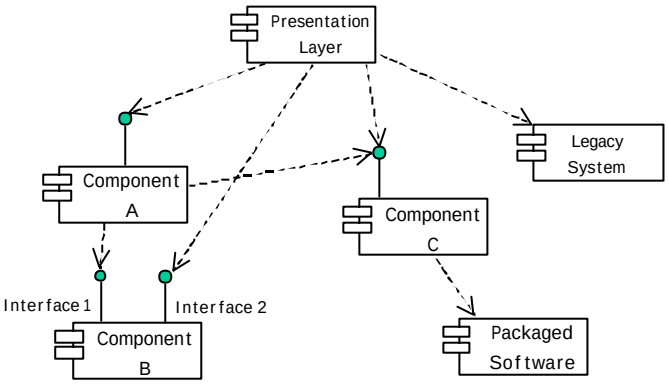


Figure 2: A Component Architecture Diagram showing six architectural units, three of which are components. For components, the interfaces can also be depicted, so usage dependencies on specific interfaces can be documented.

In a component architecture, usage dependencies are defined between the architectural units. Usage dependencies typically denote that one architecture unit calls the services of the other; so if component A depends on component B, it invokes operations of an interface on component B. The interfaces offered by components may be diagrammed: figure 2 shows component B offering two interfaces.

UML Diagrams

The process will be illustrated using an examples drawn in UML notation [1]. Figure 3 summarizes the process steps, and indicates which UML diagram is used.

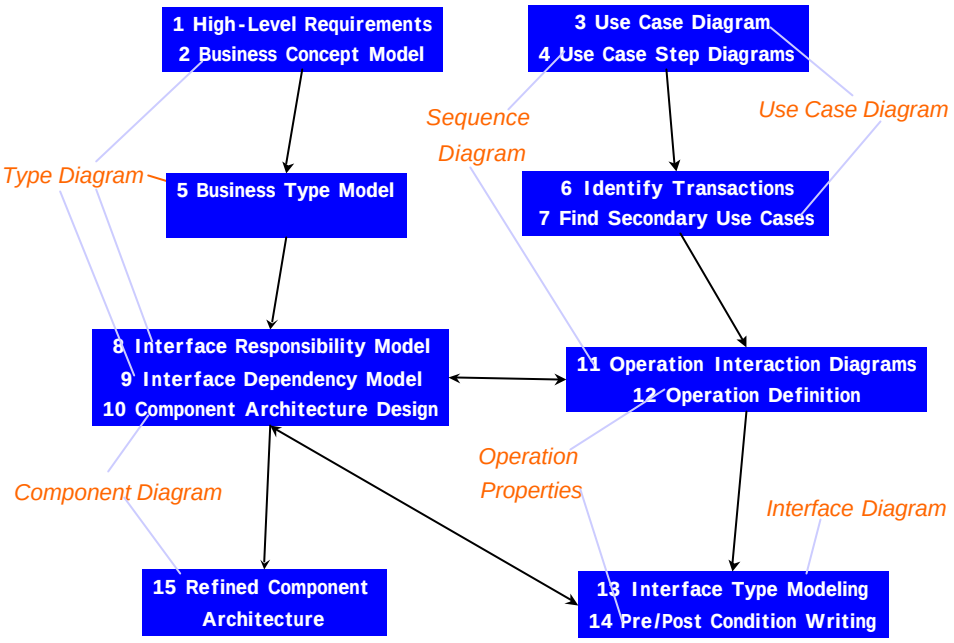


Figure 3: Steps and Diagrams

A type diagram is a special form of class diagram, in which all the classes are types. That is, they are conceptual or specification constructs, not programming language constructs. Where a tool does not provide a type diagrammer, then class diagrammer can be used, with the each class given the <<type>> stereotype.

An interface diagram is another special form of class diagram, which models the possible states of an interface. Once again, a class diagrammer tool can be used for this purpose, where a specialized interface diagramming tool is unavailable.

What's not covered?

There are number of useful component development techniques not referenced by this article, which likely warrant their own articles, and so are simply brought to the readers attention here. This paper focuses on designing a component *specification* architecture, but some projects will benefit from also building a separate component *object* architecture, component *implementation* architecture and component *deployment* diagram. Other techniques that have been found valuable are component architecture patterns, mechanisms for supporting referential integrity in component-based software, alternative component provisioning techniques and knowing how to organize UML packages for component modeling.

The steps that follow provide a basic approach for component architecture design, which can be embellished with these additional techniques according to circumstances.

Step 1: Capture High Level Requirements

It's important to capture the overall expectations of the project sponsors from the outset, to avoid the project slipping into misdirected effort. The requirements are simple testable statements that place limits on the project, register the sponsor's

expectations, and offer criteria that help decision-making. We find it useful to group the high-level requirements statements into three types:

- Functional E.g. The new application must cover the full car rental life cycle. It does not cover the vehicle life history.
- Technical E.g. The application must be built from components, that will be purchased else built-in house. The application must provide 100% data integrity
- Project Management E.g. The system-tested application must be available by dd/mm/yy.

These requirements may be adjusted as the project progresses, owing to changing business conditions or refined expectations. But if they fluctuate a lot, or changed requirements are not communicated, the project is heading for disaster.

Step 2: Build a Concept Model

The concept model identifies the principal concepts within the project scope, as perceived by the business personnel (hereafter called "the users"). The concepts and connections between concepts are documented in the form of a type diagram, as shown in figure 4.

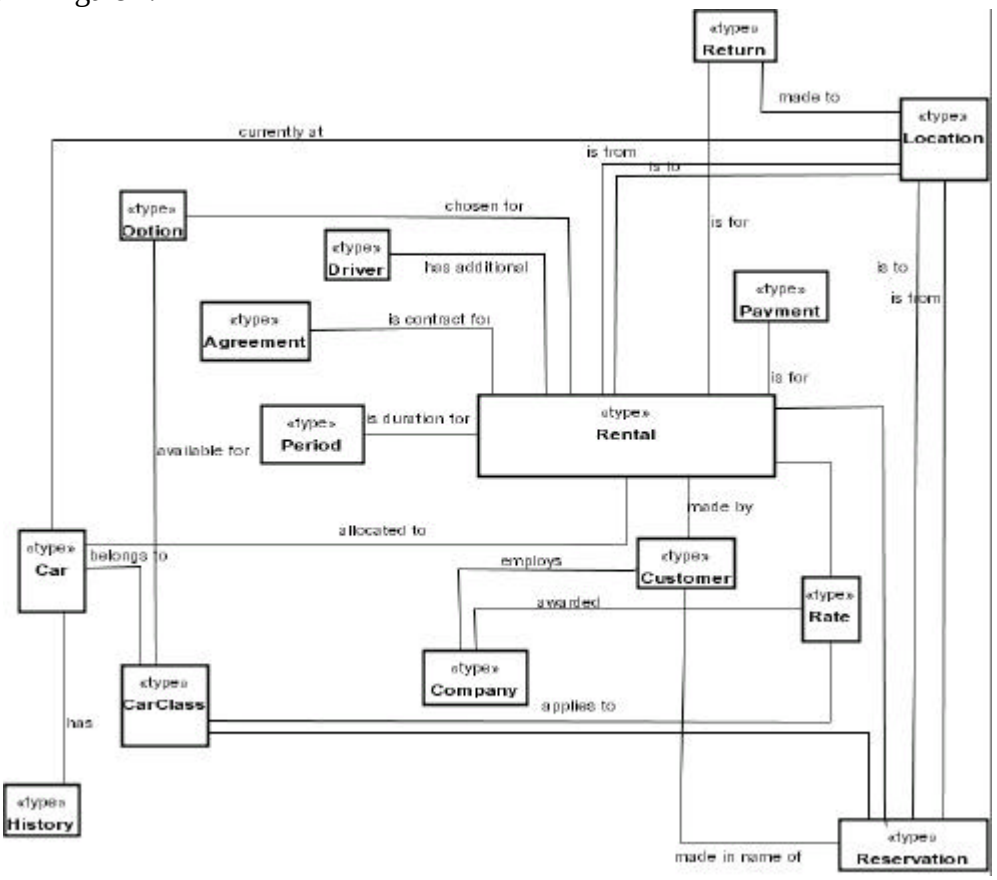


Figure 4: Users define the concepts and draw connections between the concepts. This is the beginning of the Car Rental example.

The connections are not given multiplicity or role names. The reason they are connected can be shown as an association label. Each concept should have a definition, so the model acts as a dictionary of terms.

This model is informal, and should be built by the users themselves. The concepts may be activities or things. Some may be documents; others might business objects or attributes even.

Some analysts may prefer to jump straight to the more formal business type model as described in step 5. But this informal "mind map" of business domain notions will provide valuable user input and involvement.

Step 3: Identify Use Cases

The users need to identify the actors (users of the application) and use cases (the activities they expect to perform with the new application). The finding are documented as collection of use case diagrams. An example appears in figure 5.

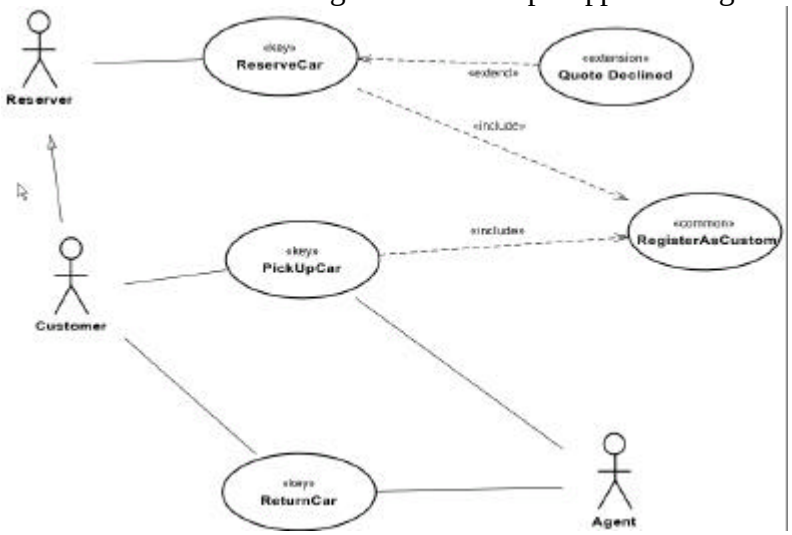


Figure 5: Three actors and three key use cases are identified. During step 4, the extension and common use cases are added to the diagram.

Use Case analysis has an extensive literature [2, 3[1]], so some brief advice only:

- First, aim to find all the key use cases: those which are central to this application, those which are uppermost in the users' minds, those which are most frequently performed, and any particularly complex use cases.
- Then try to complete the collection of things the users say they need to do with the system or which have become evident through workflow analysis of the business processes covered by the application.
- Decompose these use cases until they are "atomic" – meaning they can't be decomposed further and still be genuine use cases. Genuine use cases should support some *business* task that achieves a useful business outcome; there must always be a business event that is the motive for performing the use case (an external request arriving, a special time or date or interval occurring, or some internal business situation has been detected, which the company must monitor).
- Several actors may be involved in a use case, but only one of them initiates the use case. Actors are roles, so individuals may play several roles. Actors are not limited to human users.

- Figure 5 includes an example of a common use case and an extension use case, indicated by the <<include>> and <<extend>> dependencies respectively. These two flavors of "sub-use case" may emerge while defining the use case steps in Step 4.

Step 4: Define Use Case Steps

Every use case is examined with the users' assistance, to determine the set of exchanges (message pairs) required between the actor and the computer system, to accomplish a normal, successful execution of the use case.

Have a particular user interface style in mind when doing this (for example, web or block-mode screen or ATM), since this can considerably affect the number and nature of the exchanges. Don't try to design windows or web pages yet. The final interaction design may eventually vary from this initial "requirements gathering" exercise.

Document the steps as a series of numbered sentences, as shown in figure 6.

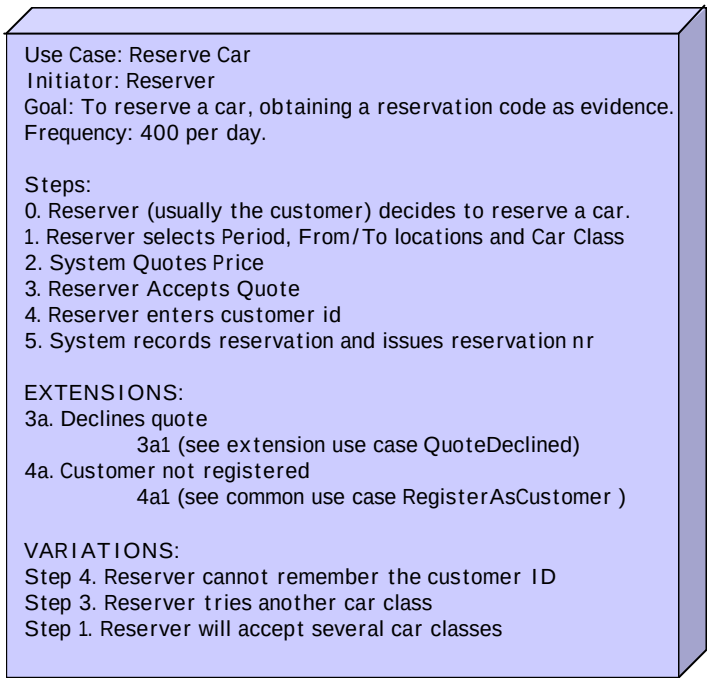


Figure 6: The documentation for the use case called Reserve Car.

Observe that more than the steps are recorded [4]. It is suggested that:

- The initiator's *goal* is identified and recorded: what the initiating actor must achieve to consider the outcome successful.
- The *business event* that is the trigger for performing the use case (immediately or with some time lag) is documented.
- The *frequency* of the use case: the number of times it is expected to be executed per month.
- Additional steps, which are only performed if some special condition arises, are documented as *extension* steps. For each extension, define the condition and the main step in which it can occur; then list the additional steps, or provide a reference to an extension use case.

- *Variations* are more informal than extensions, and merely indicate other possible courses for the use case.
- Where there are a number of significantly different paths that the use case can take then document these as separate use case scenarios –- especially for the key use cases.

Next, depict the steps of the use case on a sequence diagram. This diagram shows the actors and the application as roles (columns), and the steps as message pairs. An example is provides in figure 7. The application is the column labeled Presentation Layer. The rightmost interface column, labeled Business Logic layer, gets added in step 6.

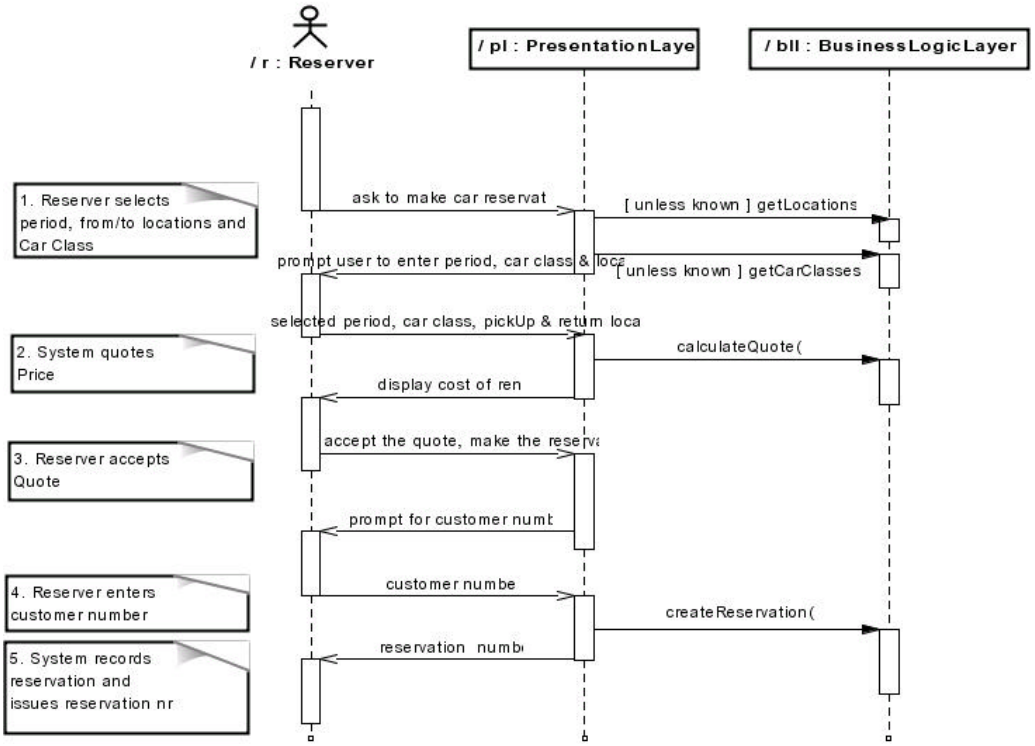


Figure 7: A sequence diagram showing the steps of the Reserve Car use case. The exchanges between actors and the user interface are defined in step 4. The operation' s calls to the business logic layer will be added in step 6.

Step 5: Prepare Business Type Model

CORE BUSINESS TYPES

A *core business type* is one of the types of thing the business has to keep track of, to run its business efficiently, legally and successfully. For example, it must keep track of its orders, it revenue, its customers, its resources and its employees. It has to keep data about each instance of these core business types, and has to give each instance a unique identifier, so its progress and state can be monitored.

The component architecture process follows the principle that the main software components should correspond to core business types. The business types of a company are relatively stable, making them a good foundation for enduring, shareable business components. The process involves identifying the business types, then assuming there will one interface per business type, and then one business component per interface – while considering whether there is any reason to vary from this heuristic at each stage.

The business type model is created by taking a copy of the concept model, and refining it. The aim is to obtain a non-redundant, normalized model of the data that the application needs to store and maintain, to properly meet the business needs. It's much like a traditional entity relationship model, except the terminology is now *types* and *associations*.

Add all the attributes you know of, label the associations (using association names or association role names if preferred), and remove attributes and associations that can be derived from others. Remove types and features that are out of scope. Use super-types (generalizations) to factor out common features, and subtypes (specializations) to document variants of types that have different features. Consider merging types that have a one-to-one association. Watch out for many-to-many associations that have attributes, and hence need to be represented as types. Redraw any association types as regular types.

Having built a satisfactory type-association model, we will pick out those types, which represent core business types, while all other types must be designated as the detail of some business type. The outcome is illustrated in figure 8. The business types identified are given the <<core>> stereotype; associations to their detailing types are highlighted using the composition annotation. So detailing types are simply a repeating or conditional or optional part of a core business type.

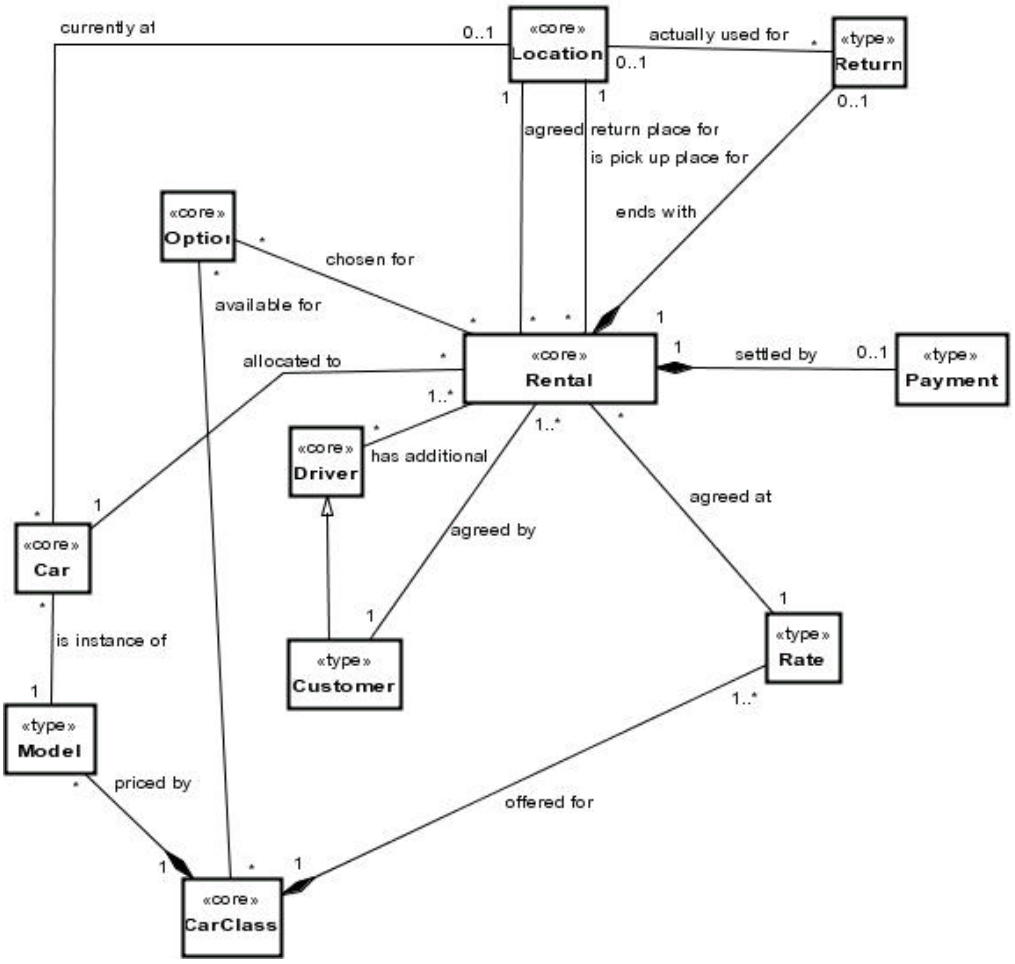


Figure 8: Business Type Model Diagram derived from the Concept Model. It was evolved into a non-redundant, normalized type-association diagram; out of scope concepts were dropped. Then the core business types and detailing types were identified.

HOW TO FIND THE BUSINESS TYPES

- Types that have no mandatory associations to anything else are core business types – or an error has been discovered in the model.
- Types that have a mandatory-one (1..1) association to another type are details of that type …
- except where all the 1..1 associations are made to classifying (core business) types. A *classifying type* primarily groups or categorizes instances of another type. For example, Car has a 1..1 association to Model (of car). Models are used to categorize Cars. Car has no further 1..1 associations to other types, so is a core business type.
- Core business types normally have an identifier that is just a single attribute -- often an artificial number or code invented by the organization.

Step 6: Identify Transactions

This step finds the business logic transactions that the *server tier* needs to support. By server tier, we mean the collection of user interface independent processing, that typically runs on a server or mainframe processor, under the aegis of an application server product or transaction processing monitor. This is in contrast to the *client tier*, which deals with the user interface appearance, inputs, outputs, dialog flow and face-value validation, that runs on a user's work station or portable device, and, in the case of HTML, on a web server.

The sequence diagram that was drawn in Step 4 is used to tackle this topic. Another interface should be added to the sequence diagram, as shown in figure 7, and this represents the business logic layer. Operations will be added to this interface that are *business logic transactions*. They are business logic in the sense that they carry out business rules, which are largely independent of any computer system. They are transactions in the ACID [8] sense: the entire transaction must be completed to ensure process and data integrity.

So after adding the interface to the sequence diagram, proceed as follows:

- Inspect each step of the use case – each actor/machine exchange.
- Decide whether the exchange requires an operation of the business logic layer to be run – to record data permanently, or to retrieve stored data, or to offer some calculation or recommendation only available from the server or mainframe or even an external service. (Generally, the presentation layer only stores data about the use case conversation itself, which is discarded at the end of the use case, and does not need to be shared by other users.)
- Given an operation is needed, first look to see if it has already been added to business logic layer for some previous use case step; if so, it can be reused.
- If the operation can't be found, try to generalize or extend some existing operation. It's better to avoid building a lot of very similar operations.
- If there is nothing to generalize, then define a new operation, to service the exchange.
- Some exchanges may utilize several operations.

- Draw the operation usage on the sequence diagram. Figure 7 shows examples of exchanges requiring one operation, two operations and no operation.

Partitioning the presentation and business logic into layers is an important architectural principle:

- The business logic layer can then be reused with a different user interface style.
- The business logic layer is normally more enduring that the presentation layer.
- Software components built for the business logic layer may be shared by several applications, which can have quite different user interface mechanisms.
- The data of the business logic layer needs to be shareable, while any presentation layer data is usually transient and for one user.

Define the parameters of each transaction, and document anything special that comes to mind about the required behavior of the transaction (as comments). It' s not essential to define the data types of the parameters at this time, but if they are known, they may as well be recorded.

Step 7: Define Secondary Use Cases

Identify any further use cases of the application that will be needed to form a complete and coherent application.

The users, with some prompting by the analysts, should have identified most of the "line of business" use cases that support the primary aims of the application. (For example, the use cases supporting the activities of an end-to-end business process.) But an examination of the business type model will usually reveal the need to maintain various codes, tables and lists that the primary use cases depend on. This may be relatively static data. In the Car Rental example, this is the case for rental locations, insurance options and the different car classes. The values might be obtainable from other systems; otherwise users will need to maintain the codes and lists manually.

Actors, use cases and steps of the secondary uses cases should be documented as in steps 3 and 4. A sequence diagram of each use case is drawn, with the actor/application exchanges depicted, and the transactions of the business logic layer identified. These use cases are often quite simple, involving the maintenance of tables of codes and their meanings, for example.

The data that can be obtained from other automated systems should be identified. This can be obtained by periodic data transfer (a use case with a non-human actor, if execution initiation is automated) or run-time calls to other systems (must be shown in the component architecture).

Step 8: Decide Interface Responsibilities

The project now has:

- a good summary of the required business logic – in the form of transactions
- a good summary of the data that needs to be stored – in the form of business types.

The project will propose a set of the *business components* that can support all this, and then decide how these components can fit together in an architecture that meets the high-level requirements – functional, technical and management. We will be aiming to reduce dependencies between components, to obtain an architecture that is easier to manage. Although the architecture will eventually contain lower-level "infrastructure" components, the process first focuses on the higher-level business components.

To achieve this end, the process first identifies the *interfaces* that the components will support, and will decide upon the components themselves in a later step. After identifying the interfaces, some critical responsibilities are assigned to the interfaces.

Here are the main actions used to identify interfaces and decide their responsibilities:

- Begin the interface responsibility diagram (IRD) by making a copy of the business type model.
- Add one interface for each core business type identified earlier, unless there is a good reason for doing otherwise. (For example, it is already decided that the existing Car Maintenance System should supply the Car operations, and these are available through two interfaces).
- Add a composition association between the interface and the core type, indicating that a single interface (instance) will manage *many* instances of the core business type. So an interface will operate on all, or possibly some sub-set of all, instances of one business type. We call this style of interface a "manager interface", and by convention name it I <business type> Manager. For example ICustomerManager or IProductMgr. If there' s a good reason for defining interfaces that manage just one instance of a business type, it is permitted – though it this not the recommended first-cut solution. Manager interfaces can perform operations like searching for all business type instances that meet a particular criterion, while instance-based interfaces cannot [5 [1]]. The resulting diagram is shown in Figure 9.

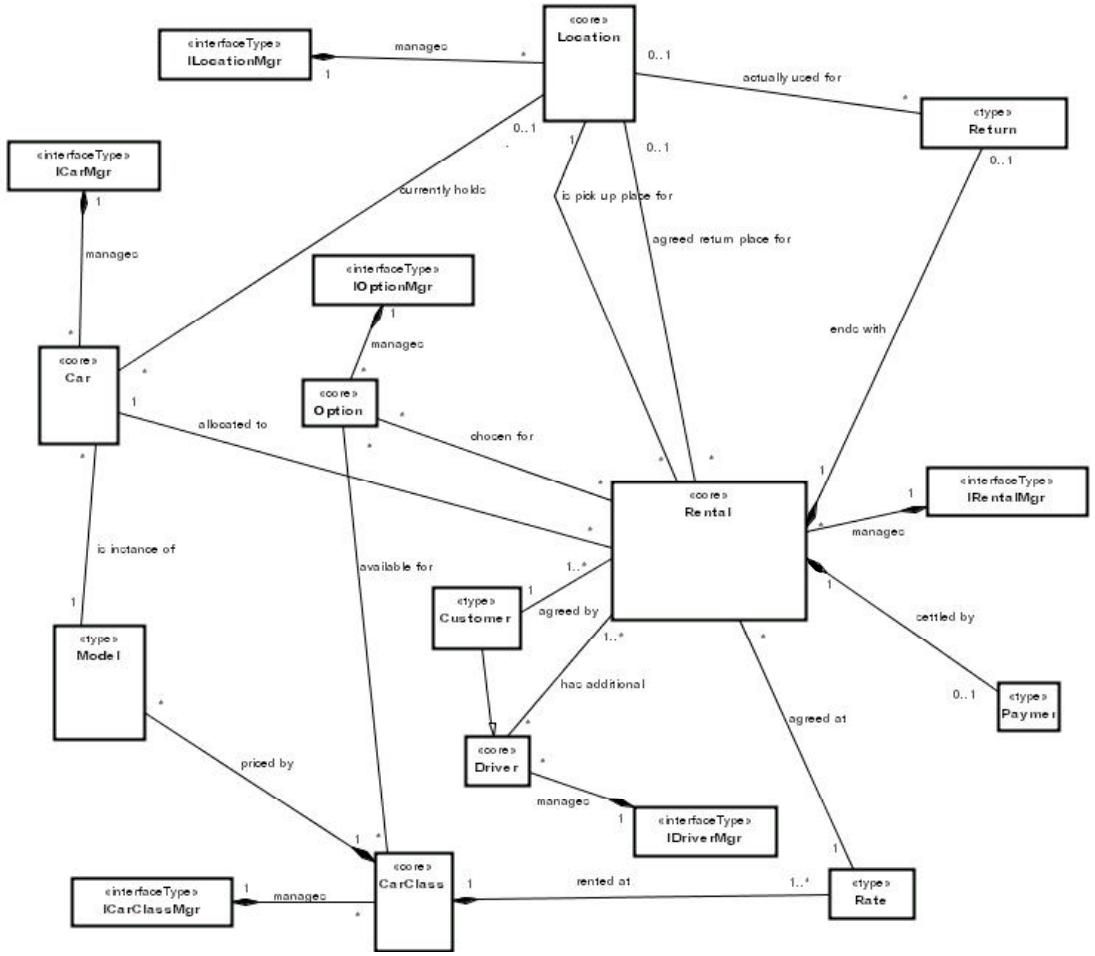
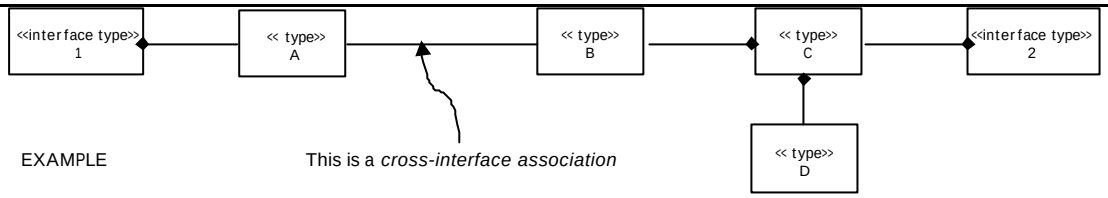


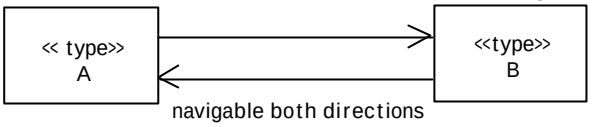
Figure 9: Initial Interface Responsibility Diagram showing one proposed interface for each core business type. The composition association adornments enable to reader to see which types each interface is responsible for. The cross-interface associations have no such adornment.

- The interface is also responsible for all the detailing types of the core business types it manages. By responsible, we mean (the implementation of the component that offers) this interface needs to provide persistent storage for all this data. (Behind the scenes, the responsibility may get delegated to some other component, but at this point, we assume this interface does it.) The interface will also offer the operations that maintain (the attributes and associations of) the core and detailing types it is responsible for – providing searches on these types, applying the business rules for these types, and so on.
- Next, consider each cross-interface association. These associations are identified by not having any composition symbol (the black diamond shape) on them. Decide in which direction each of them association should be navigable. The cross-interface associations box provides the details. The resulting diagram appears in figure 10.
- In the next step, the interface that is responsible for the integrity of this association gets decided. It can actually be different from the interface that has the responsibility for storing the association.

CROSS-INTERFACE ASSOCIATIONS



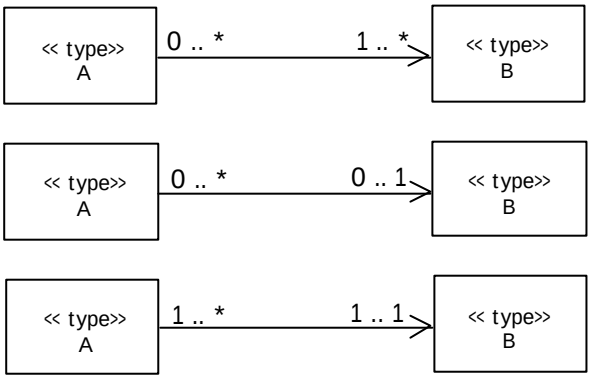
Decide whether to provide navigability in both directions, or just one direction, or possibly none. Use an arrow-head on the association line, to indicate the navigability requirement.



The decision should be based on usage. If frequent navigations are needed in both directions, then represent this on the IRD using two navigable associations.

This has an updating overhead, so architects often prefer to support navigability in just one direction, especially for highly volatile associations, or where usage is infrequent.

Since usage figures can be hard to come by, and the effort involved in collecting them is high, and because we are aiming to minimize dependencies, a simple rule of thumb is often used to decide the favored navigable direction. The type for which the association is mandatory should be favored. If both are mandatory or both are optional, then favor navigability in the "1" direction. The examples should make this clear:



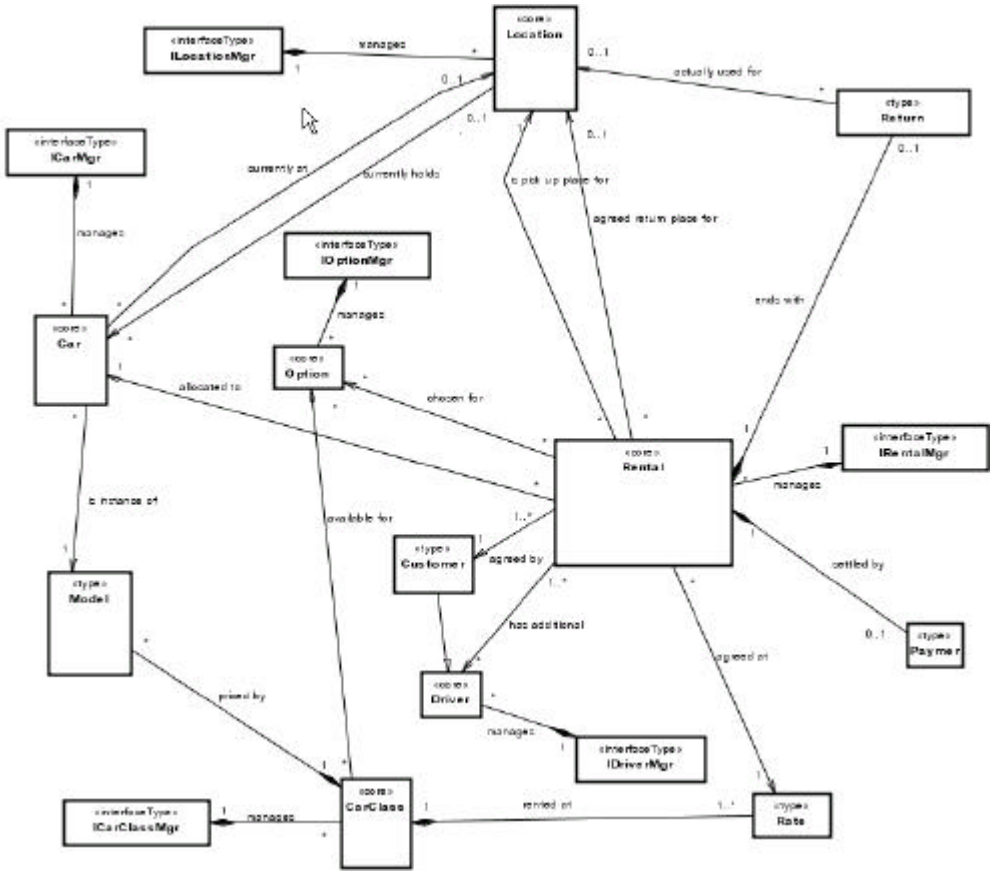


Figure 10: The Interface Responsibility Diagram now showing the proposed navigability for cross-interface associations.

Remember, interfaces are definitions of behavior, which are eventually expressed as a collection of related operations. They are not actually collections of types.

Step 9: Determine Interface Dependencies

This step investigates the usage dependencies between interfaces. These are initially derived from the Interface Responsibility Diagram. They can be derived more accurately from the Operations Interactions, once these have been studied (see Step 11).

A general architectural goal is to reduce component dependencies, which simplifies development, testing and future maintainability. It can make components easier to reuse. This can be achieved by reducing interface dependencies, or by assigning groups of interfaces which are inter-dependent to the same component.

An interface dependency diagram of the kind shown in figure 11 provides a useful summary of the interface dependencies. The architect can then attempt to adjust the responsibilities of interfaces so there are less interface dependencies, or can decide to specify components that support multiple business type-based interfaces.

So at this point, components need to be proposed, and this step will have to be performed in parallel with step 10.

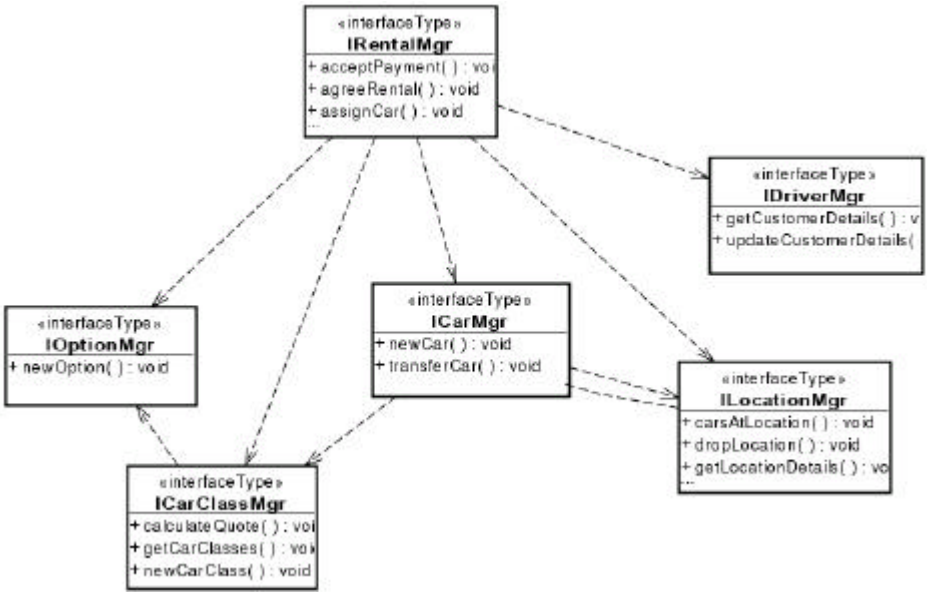


Figure 11: The initial Interface Dependency Diagram derived from the IRD shown in figure 10. Interface dependencies can be changed by altering interface responsibilities. This normally has to be done, to achieve a satisfactory component architecture.

Furthermore, two significant architectural decisions need to be made, based on the technical requirements, and on previous experience or corporate standards:

- Decide whether to allow cyclic (circular) dependencies in the architecture, or only acyclic (hierarchical) dependencies, or whether the business components should be independent of one another.
- Decide whether to include an application component in the architecture.

APPLICATION COMPONENT

An application component, if included in the architecture, processes all requests arising from the client-side of the application, and delegates the work to business components. The application component controls all the transactions, and the business component' s operations are all sub-transactional. (Without an application component, the business components offer transactional operations, which are directly invoked by the presentation layer.)

This approach can enable the business components to be relatively independent, and reusable in other contexts. The application component takes responsibility for the referential integrity of inter-business component associations. The business component might still store references to data managed by other interfaces, but the application component ensures these are valid, extant references.

If an application component is introduced, its interfaces should be included in the Interface Responsibility Diagram, and each cross-interface association should be labeled with which interface is to be responsible for its integrity.

Here is a summary of what needs to be done:

- Decide on architectural style, which might involve introducing an application component, and its interfaces.
- Assign each transactional operation of the Business Logic Layer (see step 6) to the most appropriate interface.

- Complete the interface responsibility diagram (IRD). Document which interface is responsible for the integrity of each cross-interface association.
- Draw the initial interface dependency diagram, deriving the dependencies from what is shown in the IRD: where you can navigate from A to B, then assume the interface managing A will depend on the interface managing B.
- Investigate the dependencies between the interfaces. See if they can be reduced.
- Identify a set of tentative components and examine their dependencies. Draw the component architecture diagram as described in Step 11.
- Work on both diagrams, aiming to reduce the dependencies and achieve the architectural objectives defined earlier.

Step 10: Prepare Component Specification Architecture

COMPONENT SPECIFICATION
A component specification is the complete specification of a component, which can then have alternative component implementations. The component specification is primarily a list of fully-specified interfaces, but it may have additional rules not defined in any interface, to which all implementations must conform. In UML, components correspond to component implementations or to installed components. The component specification concept is not distinct in UML.

Prepare the component specification architecture diagram for the application, depicting:

- the proposed components
- the interfaces offered by each component
- the interfaces required by each component, as component-to-interface dependencies.

The dependencies can be derived from the interface dependency diagram. A component specification can have a dependency on an interface offered by another component. (Say interface IA depends on interface IB in the Interface Dependency Diagram. And Component CA offers IA and component CB offers IB. Then the architecture diagram should depict that CA depends on IB.)

As mentioned in the previous step, the component specification diagram and interface dependency diagram, need to be developed in parallel. After selecting certain component architecture design objectives, the interface responsibilities and interface assignments are adjusted until the optimal architecture is achieved. A first-cut architecture for the Car Rental example is shown in figure 12.

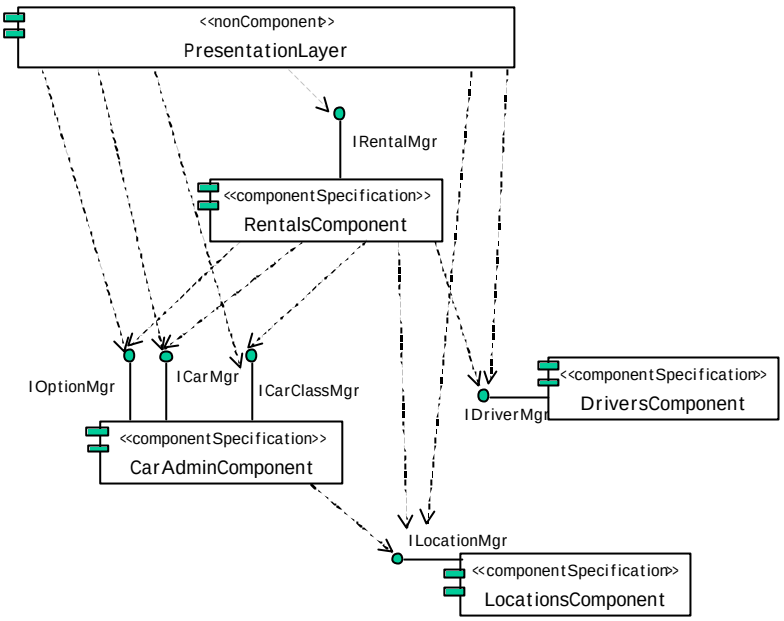


Figure 12: A Component Diagram depicting the Component Specification Architecture. The non-component Presentation Layer is shown, directly using all the business components. The component /interface dependencies are derived from the Interface Dependency Diagram. IOptionMgr and ICarClassMgr are offered by the same component as ICarMgr, partly on account of their small size, but also owing to their strong dependencies. An acyclic dependency was removed by moving allCarsAtLocation( ) from ILocationMgr to ICarMgr.

Step 11: Specify Operation Interactions

An operation often needs to call other operations, of the same or different interfaces, in order to accomplish its work. To help figure out and document these calls, sequence diagrams of operations can be drawn.

Draw sequence diagrams for selected operations, showing the interface that offers the operation (as a role) and all the interfaces it calls (as roles).

On the sequence diagram, depict which operations of other interfaces are called, and in what order, and under what conditions. If one of the called operations in turn calls other operations, including he operations of infrastructure components, or the functions of legacy systems, these can be progressively included in the diagram. Figure 13 provides an example.

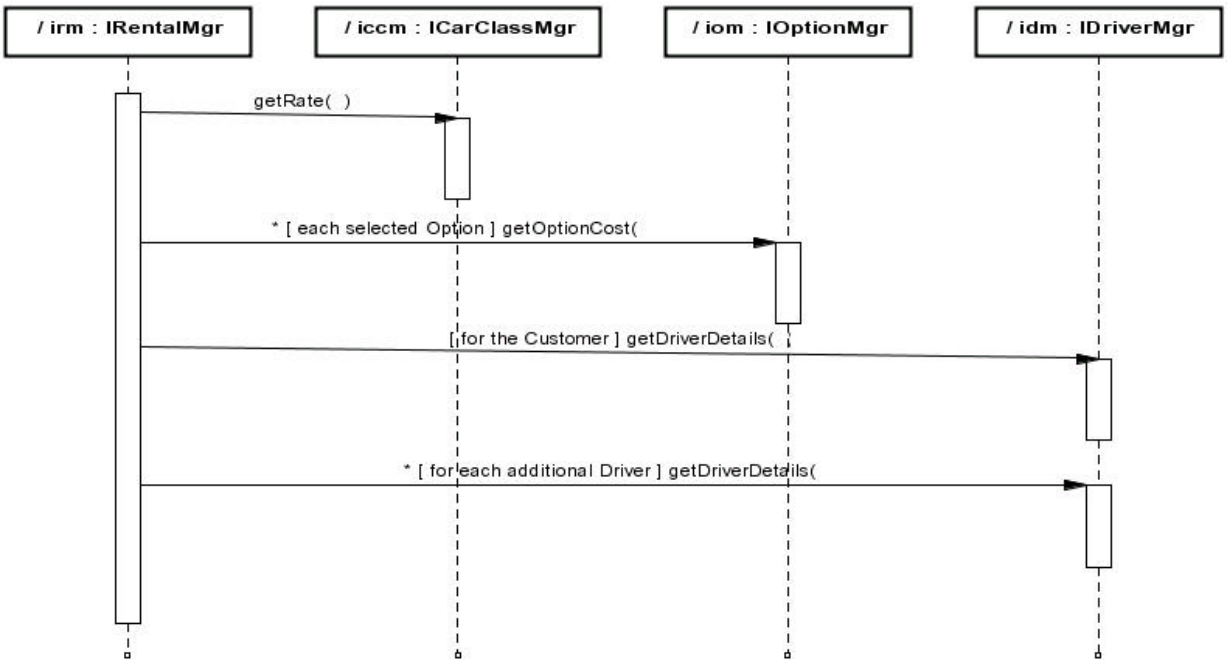


Figure 13: A sequence diagram for calculateRentalCost( ), an operation of the IRentalMgr interface. This specifies the calls calculateRentalCost( ) must make. Since this is a specification diagram, all implementations must perform these calls, though other, non-specified calls may also be made by an implementation.

It is suggested that a sequence diagram is drawn for each operation offered by the application interface, if you have decided to include one in the architecture. If you have not, it is suggested a sequence diagram is drawn only for those business interface operations which are serviced by calling other interfaces.

Remember these are specification diagrams. They state that every implementation must perform these calls. These diagrams form a part of the operation specification. If this is not the intention – it is just the preferred design for now, then it would be an implementation sequence diagram.

In practice, you may prefer to place the calls of several operations on one diagram, using annotations to make it clear which operations are being depicted.

The sequence diagrams should conform the dependency rules enshrined in the interface dependency diagram drawn earlier. An alternative approach is for the interface dependency diagram and the component diagram to be adjusted as new dependencies are discovered on the sequence diagrams. How you proceed depends on the confidence you have in the earlier diagrams. It is advisable to minimize inter-dependencies between components, so any new dependencies should be introduced with caution.

Step 12: Operation Definition

As the sequence diagrams are developed, new operations will be identified and allocated to the interfaces. Provide each operation with a meaningful name (naming conventions can help here), identify and name the parameters, and provide comments that describe any special features of the operation.

As the business components (and possibly application component) stabilize – most of the operations having been found, and the signatures stable – it is time to push on with steps 13 and 14. These involve interface type modeling, and writing pre- and post-conditions – as advocated by the Catalysis approach [7]. Together, these steps encourage the creation of a rigorous specification for each operation, that is later utilized to both code and test the component. It subsequently provides the information the developer needs to code calls to the component's operations. The developer should not need to examine the implementation code.

Step 15 completes the specification process: the initial component specifications get revised as their interfaces are detailed, or as a decision is made to reuse or adapt a pre-existing components, which may require compromises within the architecture.

Closing Remarks

The process described here has evolved during the five years or so that we have been helping organizations build industry-strength component-based business systems. It provides a systematic way of obtaining an architecture early in a project, so that project costs, schedules and organization can be addressed. In this article, we focused on the business components. When these are in place, the lower- level infrastructure components and any higher-level application components can be fitted around them.

It is intended to be a general-purpose approach, and should be applicable to the EJB, COM+ and CMM component models.

A similar process can be used for multi-application component architectures, using the key use cases of each application.

References
[1] UML 1.3, Object Management Group, www.omg.org, March 1999 .
[2] Ivar Jacobson, M Christerson, P Jonsson and G Oevergaard, Object-Oriented Software Engineering: A Use-Case Driven Approach, Addison-Wesley, Reading, MA, 1992.
[3] Craig Larman, Applying UML and Patterns, Prentice Hall PTR, Upper Saddle River, NJ,1998. [1] Chapters 6, 16 and 26
[4] Alistair Cockburn, Using Goal-Based Use Cases, Journal of Object-Oriented Programming 10 (7) Nov/Dec, 1977
[5] Peter Herzum and Oliver Sims, Business Component Factory, John Wiley, 2000. [1] pages 341-343
[6] Paul Allen and Stuart Frost, Component-Based Development for Enterprise Systems, Cambridge University Press, Cambridge, UK, 1998.
[7] Desmond D' Souza and Alan Wills, Objects, Components and Frameworks with UML: The Catalysis Approach, Addison-Wesley, Reading, MA, 1999.
[8] Theo Harder and Andreas Reuter, Principles of Transaction-Oriented Database Recovery, ACM Comp Surv. 15 (4), December 1983.

Acknowledgements

Many colleagues and customers have helped to develop and test various ideas in this article. Special thanks go to John Cheesman for his many contributions and reviewing this article.

John Dodd
Principal Technologist
Computer Associates
Chertsey, UK