ApplyinAg UML in Your Real World VFP Applications[1]

Hector J. Correa

hector@hectorcorrea.com

http://www.hectorcorrea.com

What is modeling?

In a nutshell, a model is a simplification of reality[2]. We create models because we want to create a simplified version of a particular problem or solution. Almost all engineering areas utilize some type of formal modeling. Construction engineers create blueprints of their buildings before they build the actual structures. Electronic engineers publish models detailing how radios are created so technicians can repair them. In software, engineering models are used to depict the structure and behavior of software systems.

Models used in a VFP application

Today, the modeling language of choice is the Unified Modeling Language (UML). UML can be used to specify both the structure and the behavior of a computer system.

However, VFP applications require more than just UML models. We still need entity-relationship diagrams (ERD) to represent our databases. Screen prototyping is as valuable as it has always been. The UML does not replace the techniques that we have used for years. UML is primarily used to model other areas of the system.

What about flow charts? Is it time to throw them away? No, it is not. Flowcharts are still a valuable tool, especially if you are developing a project that does not involve object-oriented programming. The UML provides a special type of flowchart called Activity Diagrams. A very common use for flowcharts is to facilitate the migration of a FoxPro 2.6 application to VFP 7. In this scenario, flowcharts can be created to depict the processes involved in the (often large) program files found in the typical 2.6 applications. From these diagrams, you can start creating other (perhaps UML) diagrams to represent an object-oriented solution to be implemented in the new system.

Why do we model?

There are two basic reasons why we model:

- To understand a problem
- To communicate with others

Since a model is a simplification of reality, we can create models of complex problems that will enable us to focus on the areas that are most important.

In general, modeling allows us to express high-level ideas much clearer than with words alone. After all, there is a reason developers say that a diagram is worth 1024 words. Modeling can help you to communicate your ideas to both your fellow developers as well as your clients.

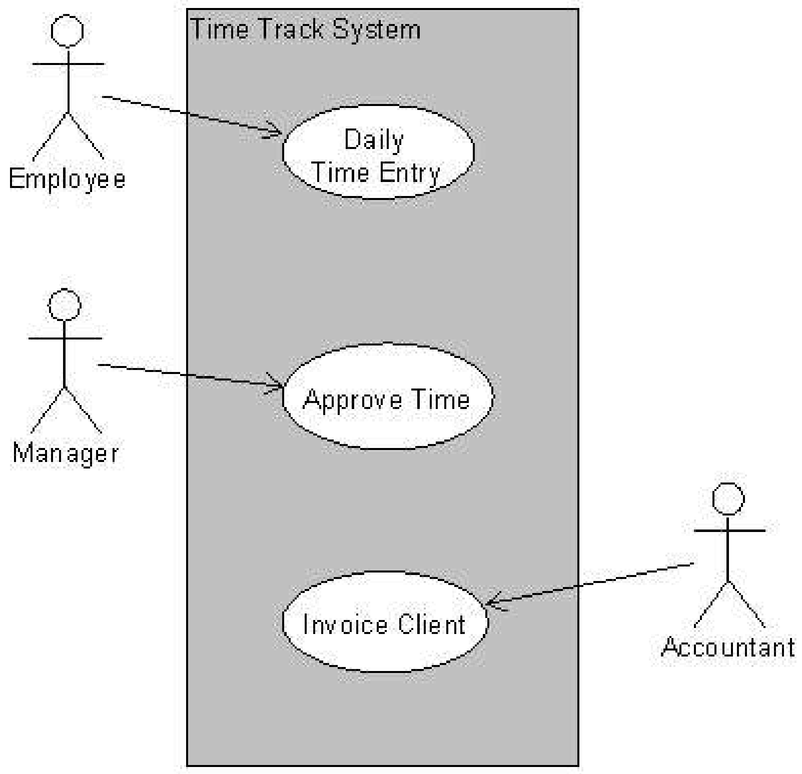
In his book, An Introduction to General Systems Thinking, Gerald Weinberg says that "contrary to popular belief, scientists use mathematics to make things clearer, not more obscure". Make sure you do the same when modeling your VFP applications. You models must help your readers; otherwise, there is no point in creating the models. In the following sections we will discuss several techniques that you can use to ensure that your models will provide value to the readers.

Who will read your model?

Before creating a model, you should ask yourself this simple question: For whom am I creating this model? The target user of your model will drive most of the type, and scope, of modeling that you need. Neglecting this step could lead to diagrams that are too complex (or too simplistic) for the readers and, therefore, fail to provide value. Typical types of readers are:

- A technical person versus a non-technical person
- An internal reader versus an external reader
- An experienced reader versus a novice reader
- Nobody else

If the reader is a non-technical person, then you know upfront that you won't need technical details on your diagram. Use Case diagrams and activity diagrams are good diagram candidates for non-technical persons. A Use Case diagram, like the one depicted below, could be used to capture the user's perspective of a Time Tracking system.

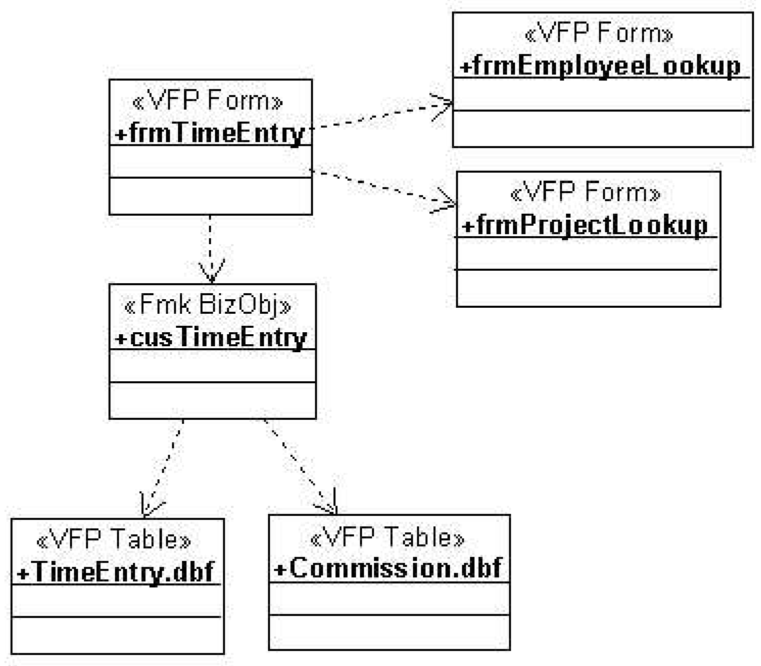


On the other hand, if the reader is a technical person, you should also define the level of experience required to read your models. If the reader is a team member that is already familiar with the way systems are built in your company, then you can probably eliminate a lot of common practices just to create a simpler image.

If the reader is a brand new developer in your company with no experience whatsoever in the class libraries used at your company, then you may need to provide a separate diagram to specify the general philosophy at your company and then another diagram to represent the problem at hand.

Even if no one else except you will read your diagram, you should keep that in mind. The level of detail required in a diagram for your own personal use is quite different from what other people will require.

The following picture shows a class diagram with the different classes that are used in a Time Entry form:



Unified Modeling Language

The Unified Modeling Language (UML) is a modeling language for writing software blueprints. A language provides the vocabulary and the rules for combining words in that vocabulary for the purpose of communication. A modeling language is a language whose vocabulary and rules focus on the conceptual and physical representation of a system.

The UML is not a software development process. A software development process tells you what models you should create and when to create them. The UML, on the other hand, will tell you how to create and read well-formed models[3].

There are nine types of diagrams in the UML. The following table summarizes these diagrams and their main usage:

UML Diagram	Type	Use to show
Use Case diagram	Behavioral	User's perspective
Class diagram	Structural	Classes and interfaces
Object diagram	Structural	Objects and their relationships
Sequence diagram	Behavioral	Message passing (focus on time ordering)
Collaboration diagram	Behavioral	Message passing (structural organization)
Statechart diagram	Behavioral	Event-ordered behavior (focus on state)
Activity diagram	Behavioral	Event-ordered behavior (focus on activity)
Component diagram	Structural	Dependencies among components
Deployment diagram	Structural	Run-time processing nodes

In this white paper, we are going to review, in detail, how to best employ Use Case diagrams and Class diagrams when modeling VFP applications.

Use Case Diagrams

Use Cases are perhaps the most confusing artifact in UML. The literature always mentions how important Use Cases are, but falls short when describing how to use them in a real world system.

When talking about Use Cases, there are five terms involved:

- Actor
- Use Case
- Use Case Diagram
- Use Case Description
- Use Case Model

An Actor is a person or another system that interacts with the system that you are modeling. Unless you are developing business-to-business (B2B) applications, in most VFP applications, nine out of ten Actors are human actors. In UML, Actors are represented by stick figures.

A Use Case is a chunk of functionality that the system offers to its Actors. In UML, Use Cases are represented by oval figures. A Use Case Diagram is used to represent the relationship between Actors and Use Cases or between Use Cases themselves. The Use Case Description is the detailed list of steps that indicates the interaction between an Actor and the system for a particular Use Case. All Use Cases, together, make up the Use Case Model[4].

Use Case diagrams are an excellent technique to represent the user's perspective of a system. They are, perhaps, the only example of a user-oriented diagram of the UML. You can use these diagrams during the initial stages of a software development process to depict the behavior that users expect from the system.

In a VFP application, Use Cases typically translate to data entry forms and reports. In other words, they translate to the pieces of the software that they are in contact with [5]. While Use Cases can be used to model all data entry forms and reports, in most scenarios they provide a lot of value when used to model the most important (and usually most complex) forms and reports of a system.

Keep in mind that there is not a one-to-one relationship between Use Cases and data entry forms. A complex Use Case might translate into several VFP forms, while a simple Use Case might translate to a single form. Also, note that not all data entry forms and reports need to be modeled as Use Cases. For trivial data entry forms a simple prototype might be all that you need.

When drawing Use Case diagrams, you can use relationships to connect actors with Use Cases or Use Cases with other Use Cases. The most common relationships are:

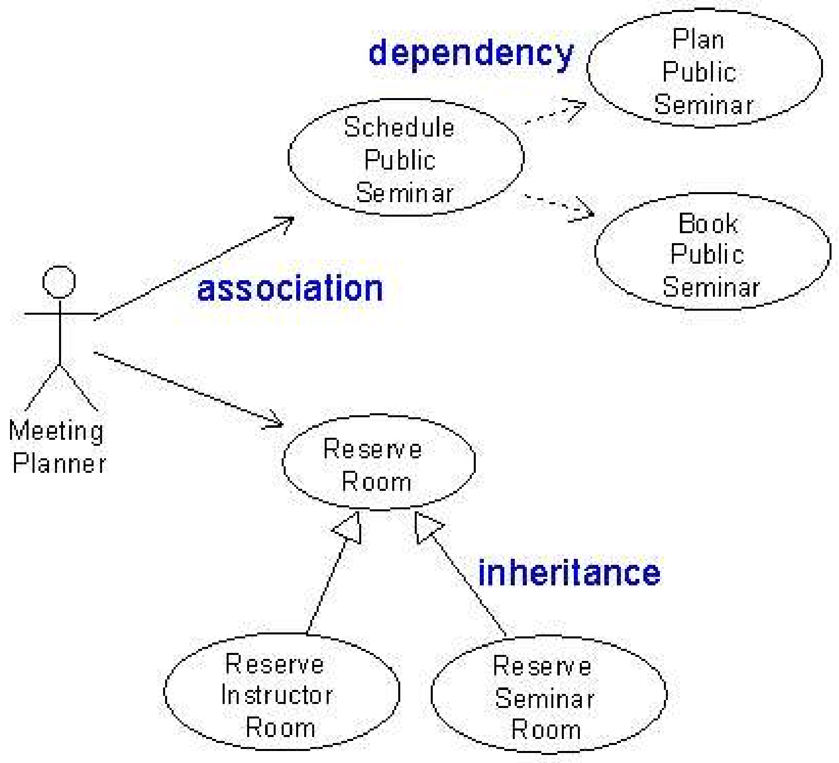
- **Association** is the relationship that is used to connect actors with Use Cases.

- **Dependency** is a special type of relationship used to connect two Use Cases. A dependency represents the fact that one Use Case relies on another Use Case. In the example depicted below, "Schedule Public Seminar" depends on two other Use Cases to carry on the planning and booking processes.

  There are two adornments that can be applied to the dependency relationship: extend and include. In theory, these two modifiers seem to be a good idea. However, in practice, they usually cause a lot of trouble and confusion among developers. It is very easy to spend hours trying to define which one is appropriated on each case. In my opinion, you should stay away from these two adornments when creating your first Use Case diagrams[6].

- **Inheritance** (a.k.a. Generalization) is another special type of relationship used to connect two Use Cases. This relationship is used to indicate that a Use Case is a variation of another one. In the example described below, the process to reserve a room for an instructor and for the seminar is basically the same, and thus, both are inherited from a single Use Case.

The following picture shows how these associations can be used in a Use Case diagram:



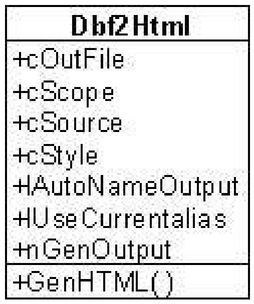
Although Use Case relationships are nice, it is important to keep in mind the reader of your Use Case diagram. If the reader of your diagram is a non-technical person, you may want to keep your diagram with a minimal number of relationships. The more technical words, like dependency and inheritance, that you add to your diagram, the less comfortable a non-technical person will feel when reading it.

Class Diagrams

The class diagram is, by far, the most popular UML diagram. Class diagrams are to object-oriented systems what Entity-Relationship diagrams are to databases. A class diagram is used to model the structure of a software system in terms of the classes that make up the system.

As opposed to Use Case diagrams, which give you the user's perspective of the system, class diagrams are used to represent the developer's perspective of the system.

In UML, a class is represented by a square shape with the class name inside. Properties and methods can be specified in additional compartments. The following picture shows how the class _Dbf2Html that comes with VFP could be diagramed using UML:



Class diagrams can be created to represent a system at several levels of detail: conceptual, logical, and physical. The UML symbols used to create any of these diagrams are always the same. The difference between these various levels of detail is in the classes modeled.

On a conceptual diagram, for example, we depict analysis classes that are usually borrowed from the problem domain, for example "Invoice".

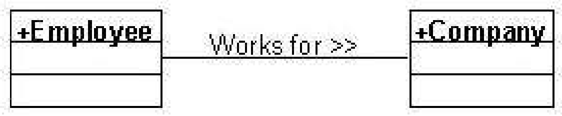
In a logical class diagram, the focus shifts to the design view of the system, and thus, we may decide to split the invoice conceptual class into two design classes like the "InvoiceHeader" class and the "InvoiceDetail" class because, from the design point of view, we know that an invoice is composed of a header and several detail lines.

A physical class diagram depicts implementation classes. In other words, the actual classes in the source code, for example, a form called, frmInvoice, contained in a SCX file or a business class cusInvoiceBizObj based on VFP custom base class and stored in a VCX file.

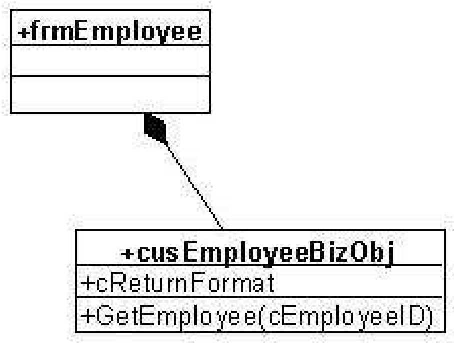
You may be wondering if you really need these three levels of class diagrams. The decision on what class diagrams you need depends, as I've said before, on why you are modeling and who is going to read your model. If you are working on a new system where the concepts are not very clear to you (or your reader), then you will probably need the three levels. However, if you are modeling a system with which you (and the readers of your model) are very familiar, then you can probably jump right to the physical layer.

There are four types of relationships that can be used in a class diagram to connect classes:

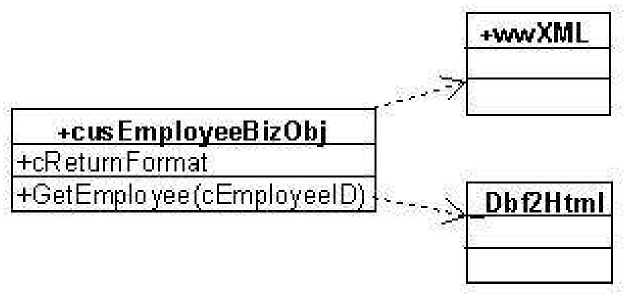
• **Association** is a structural relationship that indicates that a class is connected to another class. At its simples form, association is just a line connecting two classes, like for example to indicate than "employees work for companies" in the next picture.



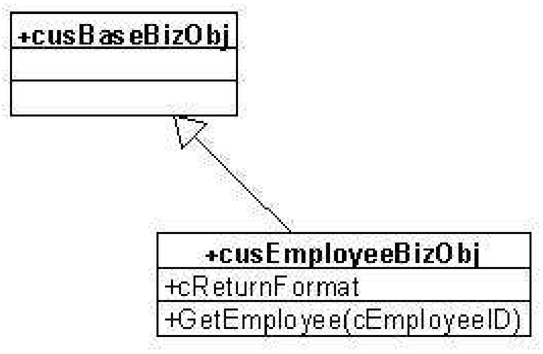
Association can be extended to express aggregation and composition between classes. The following picture shows how composition (a black diamond) can be used to indicate that an Employee's business object will be dropped inside the Employee's form.



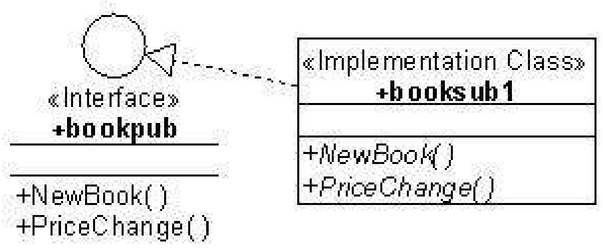
• **Dependency** relationship is used to express that one class uses another class. In a VFP application, a dependency usually represents the fact that one class will issue a call to NewObject()or CreateObject() to use another class at runtime.



• **Inheritance** (a.k.a. Generalization) relationship is used to express class hierarchies. In a VFP application, inheritance is implemented via the clause AS of the DEFINE CLASS and CREATE CLASS commands.



• **Realization** is a special type of relationship used to model interfaces and the classes that implement them. This type of relationship is particularly useful when modeling classes that will be implemented using the IMPLEMENTS clause of the DEFINE CLASS command. The following diagram shows how the interface class and the implementation class that comes with VFP 7's COM+ Services Samples will look like in UML[7].



Class diagrams provide an excellent tool to layout the structure of VFP applications. These diagrams help you define the structure of your application in a clear and concise manner so that you can communicate it effectively to other developers. If you are developing web applications or desktop applications using an n-tier architecture, class diagrams provide a great way to visually specify the responsibilities of each tier of your application. Likewise, if you are reviewing an existing implementation, class diagrams allows you to get a picture of the structure of the application without digging into hundreds of lines of code.

Models and Source Code

As I've mentioned before, modeling is a technique that helps us to create better systems. By modeling systems, we are able to understand them better and, thus, create better VFP applications.

As you create your models, keep in mind that we model to communicate and to understand systems; not to duplicate code. Even when you are modeling implementation classes, remember that your model should be a simplification of a reality and help you (or your reader) to better understand the basic concepts.

References

- Booch, Grady et all. The Unified Modeling Language User Guide. Addison Wesley, 1999.
- Cockburn, Alistair. Writing Effective Use Case. Addison Wesley, 2001.
- Egger, Markus. Advanced Object Oriented Programming with VFP 6.0. Hentzenwerke, 1999.
- Fowler, Martin. UML Distilled Second Edition. Addison Wesley, 2000.
- Jacobson, Ivar et all. The Unified Software Development Process. Addison Wesley, 1999.
- Rosenberg, Doug. Use Case Driven Object Modeling with UML. Addison Wesley, 1999.

About the Author

Hector Correa is a Software Architect in the Kansas City Area. He has extensive experience designing and building n-tier applications. Hector is a Microsoft Certified Solution Developer and holds a B.S. in Computer Engineering from Colima's Institute of Technology (Mexico). He can be reached at hector@hectorcorrea.com

---

[1] This whitepaper was originally written for Visual FoxPro (VFP) Applications. However the concepts described on it apply to most programming languages.
[2] Booch. p. 6.
[3] Booch. p. 14.
[4] Jacobson. pp. 5 and 135
[5] Although most of the use cases in a VFP application represent data entry forms and reports, in systems where users are well aware of large processes run by the application it is possible that these processes will represent use cases.
[6] In fact, you may find that you will never need them.
[7] These two classes come with VFP 7 and are typically installed in your HOME() + Samples\COM+\Events directory.