Einführung in NaturalX

Dieses Kapitel enthält eine kurze Einführung in die komponentenbasierte Programmierung, und damit einhergehend die Benutzung der NaturalX-Schnittstelle und einem speziell für diesen Zweck vorgesehenen Satz Natural-Statements.

Folgende Themen werden behandelt:

- Warum NaturalX?
- Programming Techniques

Warum NaturalX?

Auf Komponenten-Architektur basierende Software-Anwendungen bieten viele Vorteile gegenüber traditionellen Designs. Diese sind u.a. die Folgenden:

- Schnellere Entwicklung. Die Programmierer können Anwendungen schneller erstellen, indem sie die Software aus vorerstellten Komponenten zusammensetzen.
- Geringere Entwicklungskosten. Eine allgemeine Menge von Schnittstellen für Programme zur Verfügung zu haben, bedeutet weniger Arbeit bei der Integration der Komponenten in vollständige Lösungen.
- Höhere Flexibilität. Es ist leichter, Software für unterschiedliche Abteilungen innerhalb eines Unternehmens zu standardisieren, indem Sie einfach einige der Komponenten verändern, aus denen die Anwendung besteht.
- Reduzierte Wartungskosten. Im Falle eines Umstiegs auf eine neue Version ist es häufig ausreichend, einige der Komponenten zu ändern, anstatt die gesamte Anwendung ändern zu müssen.
- Einfachere Verteilung. Komponenten enthalten gekapselte Datenstrukturen und Funktionalität in verteilt installierbaren Einheiten.

Mit NaturalX können Sie komponentenbasierte Anwendungen erstellen.

Sie können NaturalX in Verbindung mit DCOM einsetzen. Das bietet folgende Möglichkeiten:

- Andere Komponenten können auf die von Ihnen entwickelten Komponenten zugreifen.
- Sie können diese Komponenten auf lokalen und/oder Remote-Servern ausführen.
- Sie können aus Natural-Programmen über Prozess- und Maschinengrenzen hinweg auf Komponenten zugreifen, die in den verschiedensten Programmiersprachen geschrieben wurden.
- Sie können Ihre vorhandenen Natural-Anwendungen mit (Quasi-) Standard-Schnittstellen ausstatten.

Das nachfolgend beschriebene Szenario demonstriert, wie ein Unternehmen aus diesen Vorteilen Nutzen ziehen kann. Ein Unternehmen führt ein Vertriebsverwaltungssystem ein, das auf einem Anwendungskonzept auf der Basis von Komponenten beruht. In dieser Anwendung sind für jede Vertriebsstelle zahlreiche Datenerfassungskomponenten vorhanden. Alle diese Vertriebsstellen benutzen

jedoch eine gemeinsame Steuerberechnungskomponente, die auf einem Server läuft. Bei Änderungen in der Steuergesetzgebung muss man nicht die Datenerfassungskomponenten an jedem einzelnen Standort ändern, sondern braucht lediglich die Steuer-Komponente zu aktualisieren. Ausserdem haben es die Programmierer leichter, weil sie sich nicht mehr den Kopf über das Programmieren von Netzwerken und die Integration von in unterschiedlichen Sprachen geschriebenen Komponenten zerbrechen müssen.

Programming Techniques

This section covers the following topics:

- Object-Based Programming
- Defining Classes
- Defining Interfaces
- Interface Inheritance

Object-Based Programming

NaturalX follows an object-based programming approach. Characteristic for this approach is the encapsulation of data structures with the corresponding functionality into classes. Encapsulation is a good basis for easy distribution. Because there are (quasi) standards for the interoperation of software components on the basis of object models, an object-based approach is also a good basis for making software components interoperable across program, machine and programming language boundaries.

Defining Classes

In an object-based application, each function is considered to be a service that is provided by an object. Each object belongs to a class. Clients use the services either to perform a business task or to build even more complex services and to provide these to other clients. Hence the basic step in creating an application with NaturalX is to define the classes that form the application. In many cases, the classes simply correspond to the real things that the application in question deals with, for example, bank accounts, aircraft, shipments etc. There is a wide range of good literature about object-oriented design, and a number of well-proven methods can be used to identify the classes in a given business.

The process of defining a class can be broadly broken down into the following steps:

- Create a Natural module of type class.
- Specify the name of the class using the `DEFINE CLASS` statement. This name will be used by the clients to create objects of that class.
- Use the `OBJECT` clause of the `DEFINE DATA` statement to define how an object of the class will look internally. Create a local data area that describes the layout of the object with the data area editor, and assign this data area in the `OBJECT` clause.

These steps are described in more detail in the section *Developing Object-Based Natural Applications*.

Defining Interfaces

In order to be useful to clients, a class must provide services, which it does through interfaces. An interface is a collection of methods and properties. A method is a function that an object of the class can perform when requested by a client. A property is an attribute of an object that a client can retrieve or change. A client accesses the services by creating an object of the class and using the methods and properties of its interfaces.

The process of defining an interface can be broadly broken down into the following steps:

- Use the `INTERFACE` clause to specify an interface name.
- Define the properties of the interface with `PROPERTY` definitions.
- Define the methods of the interface with `METHOD` definitions.

These steps are described in more detail in the section *Developing Object-Based Natural Applications*.

Simple classes only have one interface, but a class may have more than one interface. This possibility can be used to group methods and properties into one interface that belong to the same functional aspect of the class and to define different interfaces to handle other functional aspects. For example, an `Employee` class could have an interface `Administration` that contains all of the methods and properties of the administrative aspects of an employee. This interface could contain the properties `Salary` and `Department` and the method `TransferToDepartment`. Another interface `Qualifications` could contain the qualification aspects of an employee.

Interface Inheritance

Defining several interfaces for a class is the first step towards using interface inheritance, which is a more advanced method of designing classes and interfaces. This makes it possible to reuse the same interface definition in different classes. Assume that there is a class `Manager`, which is to be treated in the same way as the class `Employee` with respect to qualification, but which is to be handled differently as far as administration is concerned. This can be achieved by having the `Qualification` interface in both classes. This has the advantage that a client that uses the `Qualification` interface on a given object does not have to check explicitly whether the object represents an `Employee` or a `Manager`. It can simply use the same methods and properties without having to know of what class the object is. The properties or methods can even be implemented in a different way in both classes provided they are presented through the same interface definition.

The process of using interface inheritance can be broadly broken down into the following steps:

- Use the `INTERFACE` statements to define one or more interfaces in a copycode instead of defining them directly in the class.
- The `METHOD` and `PROPERTY` definitions in the `INTERFACE` statement do not need to contain the `IS` clause. At this point, you just define the external appearance of the interface without assigning implementations to the methods and properties.
- Use the `INTERFACE` clause to include the copycode with its interface definition in each class that will implement the interface.

- Use the METHOD and PROPERTY statements to assign implementations to the methods and properties of the interface in each class that will implement the interface.