it is essential not to defend or explain the design.

It may be harder to prototype embedded system interfaces than standard applications for the latest GUI. If you don’t have a working prototype or a serviceable mockup, you may be able to create a serviceable simulation in software. Windows or another GUI on a PC or workstation with a touch screen can “become” almost anything you want it to be and, with the help of a little imagination, can do a fairly good job of simulating most interfaces. If nothing else, imagine yourself as the user and walk yourself through a complete scenario in your mind.

If a prototype or mockup is available, developers should try to use it for actual work, to solve real problems not just briefly, but repeatedly over an extended period of time. Don’t just measure and record one signal, use the thing to measure and record the signals at every accessible point on some circuit board. Don’t just make the robot arm move from here to there. Spend a day programming and reprogramming it to reorganize your desk. If you don’t have a prototype or simulation of the system you are developing, use a competitor’s product or an earlier version of the same line to study. Along the way you are likely to learn some of the important things you do not want to do on the new version.

The most important thing to keep in mind is to actually look at the interface. Even if you are “just a programmer,” insist on seeing a mockup or pictures of the actual hardware interface. Don’t be content to be told to “read this signal and send an on-code to that indicator.” Know what these mean and how they are arranged in relation to each other.

If you keep your eyes open and your mind on basic principles of user interface design, you can help design better user interfaces for embedded systems.

REFERENCES


Larry Constantine is a pioneer of the structural revolution and the developer of structured design. A consultant and frequent lecturer on CASE and the integration of object-oriented development with traditional structured methods, he also teaches new approaches to managing and organizing design teams and work groups. With Ed Yourdon, he wrote Structured Design, the definitive text on the subject still widely used after 14 years in print. He is a graduate of Sloan School of Management at MIT and has served on the faculties of four major universities.
FROM EVENTS TO OBJECTS: USE CASES AS THE BRIDGE TO EMBEDDED SYSTEMS

The concept of a use case can help designers develop better object-oriented solutions for embedded systems applications. Use cases are abstractions of interrelated events or interaction sequences. By analysing applications in terms of use cases and organizing the software functionality on the same basis, separating functionality into three distinct types of object modules, software can be made more robust, with components that are both more reusable and more amenable to change.

Object-oriented development methods offer many advantages for embedded systems developers. Well-structured object-oriented systems can be very robust, standing up well to the demands of changing versions and expanding system capabilities without extensive reprogramming. Appropriately specified object components can often be reused from version to version and project to project in a given application area. With careful attention to abstraction and generalization, a growing suite of reusable components can be amassed to simplify future embedded systems development.

The key to all such benefits is identifying the appropriate and well-structured set of components. If these component parts contain too much or too little, if their features are too specific or too limited, or if they are dependent on each other in inappropriate ways, they will be less usable in new or altered contexts, and the potential gains of object orientation will not be realized.

Now that it is recognized that object-orientation is no magic bullet that will slay the software development demon, increasing attention is being paid to the methods by which object-oriented systems are developed. In many, if not most cases, the use of a systematic development method will have more impact on the success of a project than the choice of programming language or even the decision to use a object-oriented versus a function-oriented software architecture. Unfortunately, many methods for object-oriented analysis and design are relatively new and unproved, while others are little more than reworkings of data modeling methods based on entity-relationship models. These latter may have the advantage of an established history of use, but applying them to object-oriented development on embedded systems poses problems. Not only is there relatively little experience or evidence for success with these entity-based models in their object-oriented reincarnations, but it is not always clear how well these models suit the realities of embedded systems.

An ideal object-oriented method for designing embedded systems would have a proven track record and be suited to the special features of embedded systems applications. Use case-driven design (Jacobson, Christerson, Jonsson, and Overgaard, 1992) may be such an approach. Originally growing from work on switching systems and telecommunications applications, it has been in use as an evolving method for development of complex object-oriented systems for over two decades. Central concepts in this method make it well suited to embedded systems applications.

Embedded Systems

There are many ways to characterize the special features of embedded systems applications and much disagreement about just how unique these may be to embedded systems. Plauger (1989) has even argued that embedded systems are qualitatively indistinguishable from other categories of software applications. If there are important differences, they are likely to be matters of degree and emphasis rather than differences in kind. As a rule, embedded systems involve close interaction with real-world physical components and mechanisms. This interaction takes the form of control, coordination, or monitoring of actual electronic or electromechanical devices. What stands out is both the importance of real-world objects and the sequence of events or transactions taking place between the embedded software and these external physical objects. These features suggest that analysis and design methods that are both object-oriented and that focus on events and transactional sequences may have some special advantages in developing embedded systems software.

Functions and Objects

An embedded system for a given application must ultimately exhibit the same effective behavior and external characteristics regardless of how the internal program is realized. Whether the features and capabilities are embodied as discrete functional units or into object classes or are merely thrown together into a collage of code, the observable performance must conform to requirements. The differences are in the packaging or organization of the requirements, that is, how they are mapped into code (Constantine, 1989; 1990).

In traditional function-oriented software architecture, the total capability of the system is organized into a hierarchy of small functional modules—routines, subroutines, or functions—that operate on data, transforming it from one form to another and carrying out the sequence of steps required for the system behavior (Yourdon and Constantine, 1979). Data itself may be organized into tables, arrays, or other structures and may be localized in specific modules, shared among a group of them, or globalized in common data areas or data bases. In any case, data and function are to some greater or lesser degree treated as separate issues and the relationship between functions and the data they operate upon is not an important or central concern in the design and programming process. In some cases data may be thought of as attached or assigned to a function or functions; in others it is regarded as apart from functions altogether.

In an object-oriented software architecture, the system as a whole is organized around data
FROM EVENTS TO OBJECTS: USE CASES AS THE BRIDGE TO EMBEDDED SYSTEMS

The concept of a use case can help designers develop better object-oriented solutions for embedded systems applications. Use cases are abstractions of interrelated events or interaction sequences. By analysing applications in terms of use cases and organizing the software functionality on the same basis, separating functionality into three distinct types of object modules, software can be made more robust, with components that are both more reusable and more amenable to change.

Object-oriented development methods offer many advantages for embedded systems developers. Well-structured object-oriented systems can be very robust, standing up well to the demands of changing versions and expanding system capabilities without extensive reprogramming. Appropriately specified object components can often be reused from version to version and project to project in a given application area. With careful attention to abstraction and generalization, a growing suite of reusable components can be amassed to simplify future embedded systems development.

The key to all such benefits is identifying the appropriate and well-structured set of components. If these component parts contain too much or too little, if their features are too specific or too limited, or if they are dependent on each other in inappropriate ways, they will be less usable in new or altered contexts, and the potential gains of object orientation will not be realized.

Now that it is recognized that object-orientation is no magic bullet that will slay the software development demon, increasing attention is being paid to the methods by which object-oriented systems are developed. In many, if not most cases, the use of a systematic development method will have more impact on the success of a project than the choice of programming language or even the decision to use a object-oriented versus a function-oriented software architecture. Unfortunately, many methods for object-oriented analysis and design are relatively new and unproven, while others are little more than reworkings of data modeling methods based on entity-relationship models. These latter may have the advantage of an established history of use, but applying them to object-oriented development on embedded systems poses problems. Not only is there relatively little experience or evidence for success with these entity-based models in their object-oriented reincarnations, but it is not always clear how well these models suit the realities of embedded systems.

An ideal object-oriented method for designing embedded systems would have a proven track record and be suited to the special features of embedded systems applications. Use case driven design (Jacobson, Christerson, Jonsson, and Overgaard, 1992) may be such an approach. Originally growing from work on switching systems and telecommunications applications, it has been in use as an evolving method for development of complex object-oriented systems for over two decades. Central concepts in this method make it well suited to embedded systems applications.

Embedded Systems

There are many ways to characterize the special features of embedded systems applications and much disagreement about just how unique these may be to embedded systems. Plauger (1989) has even argued that embedded systems are qualitatively indistinguishable from other categories of software applications. If there are important differences, they are likely to be matters of degree and emphasis rather than differences in kind. As a rule, embedded systems involve close interaction with real-world physical components and mechanisms. This interaction takes the form of control, coordination, or monitoring of actual electronic or electromechanical devices. What stands out is both the importance of real-world objects and the sequence of events or transactions taking place between the embedded software and these external physical objects. These features suggest that analysis and design methods that are both object-oriented and that focus on events and transactional sequences may have some special advantages in developing embedded systems software.

Functions and Objects

An embedded system for a given application must ultimately exhibit the same effective behavior and external characteristics regardless of how the internal program is realized. Whether the features and capabilities are embodied as discrete functional units or into object classes or are merely thrown together into a collage of code, the observable performance must conform to requirements. The differences are in the packaging or organization of the requirements, that is, how they are mapped into code (Constantine, 1989; 1990).

In traditional function-oriented software architecture, the total capability of the system is organized into a hierarchy of small functional modules—routines, subroutines, or functions—that operate on data, transforming it from one form to another and carrying out the sequence of steps required for the system behavior (Yourdon and Constantine, 1979). Data itself may be organized into tables, arrays, or other structures and may be localized in specific modules, shared among a group of them, or globalized in common data areas or data bases. In any case, data and function are to some greater or lesser degree treated as separate issues and the relationship between functions and the data they operate upon is not an important or central concern in the design and programming process. In some cases data may be thought of as attached or assigned to a function or functions, in others it is regarded as apart from functions altogether.

In an object-oriented software architecture, the system as a whole is organized around data
abstractions with functions being attached to and “encapsulated” with the data upon which they operate. Done correctly, all and only those functions operating on a specific and well-defined piece of data are combined with that data to form an object module (Page-Jones, Constantine, and Weiss, 1990). As programmed, such an object module consists of the description of the data and the descriptions of the functions operating on the data. Only these functions that form a part of the definition of the object have access to and are aware of the structure of the data with which they are combined. Any other part of the code does not see the data itself, but instead see only these functions or operations, which are used to access and operate upon the encapsulated data. These functions are often referred to as the “methods” of the object.

Any part of the program needing to make use of an operation sends a message to an object that invokes or calls for the performance of some method. A list of waiting customers might be sent a message to display itself or an elevator object might be sent a message to move to a designated floor. The client part of the program need know nothing of the internal structure of the object to which it sends a message nor anything of how the method or operation is carried out. This facilitates both development and modification of discrete pieces of the system as independent component parts.

Objects in an object-oriented software architecture often stand in for or represent things in the external world of the application. Thus an elevator object class models some of the relevant behavior of elevators in the external world. The software object can be told to start, stop, open its doors, and the like, corresponding to operations of the real physical elevator in some building. In part, it is this correspondence between software component parts and actual physical objects in the real world that makes object-orientation appealing for embedded systems applications.

Entities and Events

The basis of many object-oriented analysis and design approaches is a model of the conceptual structure of the application area. This model identifies classes of objects corresponding to actual tangible physical objects or to discrete constructs or ideas used in describing the application area. This model may sometimes be called an entity-based model, but to avoid confusion with entity-relationship modeling, it is also sometimes called a domain model. It models the content and structure of the application domain.

Examples of domain entities in an airline reservation application include such things as “passenger,” “airplane,” and “ticket,” which correspond to tangible objects in the real world, as well as conceptual entities, such as “itinerary” and “reservation.” Each domain entity is defined by its attributes, such as passenger name, issue date, ticket number, etc., for a ticket. The domain model also defines the relationship among domain entities. A passenger “has a” ticket and “flies on a” plane.

Events form an alternative dimension of applications from which object-oriented software architectures can be derived (Page-Jones and Weiss, 1989; Weiss and Page-Jones, 1992).

Events are occurrences in the environment that take place at a particular time or over time. Paraphrasing the movie, “Grand Canyon,” we might say, “Events happen!” A complete event consists of a trigger, or stimulus, some action or actions performed by the system in response to the stimulus, and some response or results returned to the “outside” or reflected in some changed state or status. An event model is a model of stimulus-action-response within a system.

The trigger or stimulus is the signal that tells the system that an event or class of events has occurred. The system must recognize the event from incoming stimuli and carry out appropriate actions, producing the correct results or output. For example, an embedded program for a process controller may get a signal from a control keypad that a button has been depressed. It must decode which button has been depressed, then, based on the particular button and the current state of the process, send the signals to open or close a valve and display the updated status on a panel.

Object-oriented software can be organized around entities or around events, and the choice of architecture can affect what kinds of changes or extensions will be easy to make and which ones will be more difficult (Constantine, 1990). Some experience suggests that object-oriented architectures organized by events or event sequences may be more robust in standing up to changing requirements of the sort that typically affect real software (Weiss and Page-Jones, 1990; Jacobson, Christerson, Jonsson, Overgaard, 1992).

Any object-oriented analysis approach must at some stage or another identify all the requisite behavior of the system and the conceptual or physical entities in the application domain. Ultimately, in a complete and purely object-oriented design, all behavior associated with all events or event sequences and all attributes and information about system state must be allocated to object modules, whether these correspond to domain entities or are concocted in the design process for other purposes.

Object-oriented design can be thought of as a problem of getting several disjoint descriptions of a problem together so that behavior is allocated to objects in a way that facilitates reuse and extensibility. One can start with different models and focus on various issues, but the goal is always this combined description.

Actors and Use Cases

Use case driven design starts with an external view of the system to be designed. It does not start with the conceptual content of the application, as in entity-oriented methods, but with an analysis that is user oriented and usage oriented. The first questions are: Who or what is going to use (interact with) this system? What roles do they play in relation to the system? What kinds of uses do they make of the system. Two concepts are key in this approach: actors and use cases. These two concepts comprise an external view of the system, defining what exists outside the system (actors) and what behavior the system should be able to exhibit (use cases).
abstractions with functions being attached to and "encapsulated" with the data upon which they operate. Done correctly, all and only those functions operating on a specific and well-defined piece of data are combined with that data to form an object module (Page-Jones, Constantine, and Weiss, 1990). As programmed, such an object module consists of the description of the data and the descriptions of the functions operating on the data. Only these functions that form a part of the definition of the object have access to and are aware of the structure of the data with which they are combined. Any other part of the code does not see the data itself, but instead see only these functions or operations, which are used to access and operate upon the encapsulated data. These functions are often referred to as the "methods" of the object.

Any part of the program needing to make use of an object sends a message to an object that invokes or calls for the performance of some method. A list of waiting customers might be sent a message to display itself or an elevator object might be sent a message to move to a designated floor. The client part of the program need know nothing of the internal structure of the object to which it sends a message nor anything of how the method or operation is carried out. This facilitates both development and modification of discrete pieces of the system as independent component parts.

Objects in an object-oriented software architecture often stand in for or represent things in the external world of the application. Thus an elevator object class models some of the relevant behavior of elevators in the external world. The software object can be told to start, stop, open its doors, and the like, corresponding to operations of the real physical elevator in some building. In part, it is this correspondence between software component parts and actual physical objects in the real world that makes object-orientation appealing for embedded systems applications.

Entities and Events

The basis of many object-oriented analysis and design approaches is a model of the conceptual structure of the application area. This model identifies classes of objects corresponding to actual tangible physical objects or to discrete constructs or ideas used in describing the application area. This model may sometimes be called an entity-based model, but to avoid confusion with entity-relationship modeling, it is also sometimes called a domain model. It models the content and structure of the application domain.

Examples of domain entities in an airline reservation application include such things as "passenger," "airplane," and "ticket," which correspond to tangible objects in the real world, as well as conceptual entities, such as "itinerary" and "reservation." Each domain entity is defined by its attributes, such as passenger name, issue date, ticket number, etc., for a ticket. The domain model also defines the relationships among domain entities. A passenger "has a" ticket and "flies on a" plane.

Events form an alternative dimension of applications from which object-oriented software architectures can be derived (Page-Jones and Weiss, 1989; Weiss and Page-Jones, 1992).

Events are occurrences in the environment that take place at a particular time or over time. Paraphrasing the movie, "Grand Canyon," we might say, "Events happen!" A complete event consists of a trigger, or stimulus, some action or actions performed by the system in response to the stimulus, and some response or results returned to the "outside" or reflected in some changed state or status. An event model is a model of stimulus-action-response within a system.

The trigger or stimulus is the signal that tells the system that an event or class of events has occurred. The system must recognize the event from incoming stimuli and carry out appropriate actions, producing the correct results or output. For example, an embedded program for a process controller may get a signal from a control keypad that a button has been depressed. It must decode which button has been depressed, then, based on the particular button and the current state of the process, send the signals to open or close a valve and display the updated status on a panel.

Object-oriented software can be organized around entities or around events, and the choice of architecture can affect what kinds of changes or extensions will be easy to make and which ones will be more difficult (Constantine, 1990). Some experience suggests that object-oriented architectures organized by events or event sequences may be more robust in standing up to changing requirements of the sort that typically affect real software (Weiss and Page-Jones, 1990; Jacobson, Christonson, Jonsson, Overgaard, 1992).

Any object-oriented analysis approach must at some stage or another identify all the requisite behavior of the system and the conceptual or physical entities in the application domain. Ultimately, in a complete and purely object-oriented design, all behavior associated with all events or event sequences and all attributes and information about system state must be allocated to object modules, whether these correspond to domain entities or are concocted in the design process for other purposes.

Object-oriented design can be thought of as a problem of getting several disjoint descriptions of a problem together so that behavior is allocated to objects in a way that facilitates reuse and extensibility. One can start with different models and focus on various issues, but the goal is always this combined description.

Actors and Use Cases

Use case driven design starts with an external view of the system to be designed. It does not start with the conceptual content of the application, as in entity-oriented methods, but with an analysis that is user oriented and usage oriented. The first questions are: Who or what is going to use (interact with) this system? What roles do they play in relation to the system? What kinds of uses do they make of the system. Two concepts are key in this approach: actors and use cases. These two concepts comprise an external view of the system, defining what exists outside the system (actors) and what behavior the system should be able to exhibit (use cases).
To understand use cases we must understand actors. Actors act upon or with the system. The actors for a given application include whatever interacts with the system, everything that has a need to exchange information with it. An actor is not a user, however, but a particular role which some user can play in relation to the system. A user is an actual physical entity: any person or device or other system that makes use of the system. An actor is a role that some user may take on in relation to the system. Thus “actor” is a functional concept related to what and how a system may be used, not a class of objects in the real world.

For example, in a materials handling system in a warehouse the human operator (a user) may function in the role of a dispatcher, directing that goods go to a certain destination, or as a system administrator, setting up the codes that control access to the system. In a sense, an actor represents a class of users. A particular user, acting in a particular role in relation to the system, is regarded as an instance of that particular kind of actor. A given user may function in more than one role and a given role may be played at various times by various users. The notion of an actor, or user role, is a functional one tied to the actual required performance of the system.

A use case (that is, a case of usage) is a specific way for an actor to use a system to accomplish some purpose meaningful to the actor, that is, to a user in that role. An actor can perform a number of different things with the system, participating in various ways in the operation of the system. A use case covers a complete and well-defined interactive sequence or dialogue with the system. A use case is, thus, an abstraction or aggregation of more elementary events.

For example, a bank customer may be an actor using an Automatic Teller Machine (ATM). The entire sequence of interaction involved in a withdrawal is a use case of the ATM system, including entering the PIN, entering the amount, removal of money, etc.

The collection of all use cases covers the complete functionality of the system to be built as viewed from the outside. The system must be able to perform everything described in the use case model.

In use case driven design, the entire system architecture is governed by what users want to do with the system. Because the model is relatively easy to understand and is formulated from the user perspective, it can be used to verify with users and customers the correctness of the model. It is often desirable to support the use case model with models of the actual user interface corresponding to each use case. Interface prototypes can simulate use cases for the users by showing them what will be seen when executing the use case in the system to be built.

We will call the complete model of users, user roles, and use cases, the usage model. The usage model is typically supplemented by a more conventional domain model that represents entities or constructs from the problem domain and their interrelationships. This serves to define the “domain of discourse” between user/client and developer. The usage model is expressed in terms of the domain model. The domain model itself may take the form of an entity-relationship diagram or an object-oriented extension, such as in the Extended Uniform Object Notation (Henderson-Sellers, Edwards, and Constantine, 1992).

Entity, Interface, and Control Objects

If our systems are to have robust architecture that responds readily to new functional needs in terms of expanded or changing users, user roles, and usage, the architecture of the software should reflect the essential structure of our usage model. Unfortunately, many object-oriented methods use an entity-oriented or application domain model as the primary or exclusive basis for the design. These push the designer into allocating all behavior to objects derived from the domain model. More robust designs result from differentiating three types of object modules each with a different purpose or function in the system.

Entity objects model information in the system that must be retained over time. Usually this means information that survives from one use case to another. All behavior naturally and closely related to this information is placed within the entity object. Entity objects nearly always correspond to entities in an application domain model, that is, to tangible objects or conceptual entities within the application area. An example of an entity object in a personnel management system might be an employee with its associated data defining name, skills, employment history, etc.

Interface objects mediate between actors and the rest of the software system. They encapsulate functionality specific to particular devices or actors, thus allowing for changing interface characteristics independently of underlying functional capability and vice versa. Interface objects model behavior and information that is dependent on the interface of the system, thus, everything specifically about any interface of the system is placed in an interface object.

Control objects model behavior that is not naturally tied to any other object. Typically this behavior involves interaction with several different entities. They encapsulate policies or procedures that span multiple object classes or complex behaviors involving various classes. (Page-Jones and Weiss in their “Synthesis” method refer to these as event managers. See Page-Jones and Weiss, 1989; Weiss and Page-Jones, 1992.) An example might be a control object to encapsulate the lending policy of a college library, where this policy involves the kind of borrower, the type of material, and even the time in the semester. Another example would be the polling object in a patient monitoring system that polls and reports on various patients in different wards according to a priority scheme. In both of these examples, the behavior could be allocated to other objects (such as the borrower or the patient), but the resulting design is clumsier and less robust because the behavior does not really belong to those other specific objects.

Typically, a control object will correspond to a use case. In the actual operation of the software, a control object is instantiated for each instance of the particular use case and remains for the duration of the interactive sequence.
To understand use cases we must understand actors. Actors act upon or with the system. The actors for a given application include whatever interacts with the system, everything that has a need to exchange information with it. An actor is not a user, however, but a particular role which some user can play in relation to the system. A user is an actual physical entity: any person or device or other system that makes use of the system. An actor is a role that some user may take on in relation to the system. Thus "actor" is a functional concept related to what and how a system may be used, not a class of objects in the real world.

For example, in a materials handling system in a warehouse the human operator (a user) may function in the role of a dispatcher, directing that goods go to a certain destination, or as a system administrator, setting up the codes that control access to the system. In a sense, an actor represents a class of users. A particular user, acting in a particular role in relation to the system, is regarded as an instance of that particular kind of actor. A given user may function in more than one role and a given role may be played at various times by various users. The notion of an actor, or user role, is a functional one tied to the actual required performance of the system.

A use case (that is, a case of usage) is a specific way for an actor to use a system to accomplish some purpose meaningful to the actor, that is, to a user in that role. An actor can perform a number of different things with the system, participating in various ways in the operation of the system. A use case covers a complete and well-defined interactive sequence or dialogue with the system. A use case is, thus, an abstraction or aggregation of more elementary events.

For example, a bank customer may be an actor using an Automatic Teller Machine (ATM). The entire sequence of interaction involved in a withdrawal is a use case of the ATM system, including entering the PIN, entering the amount, removal of money, etc.

The collection of all use cases covers the complete functionality of the system to be built as viewed from the outside. The system must be able to perform everything described in the use case model.

In use case driven design, the entire system architecture is governed by what users want to do with the system. Because the model is relatively easy to understand and is formulated from the user perspective, it can be used to verify with users and customers the correctness of the model. It is often desirable to support the use case model with models of the actual user interface corresponding to each use case. Interface prototypes can simulate use cases for the users by showing them what will be seen when executing the use case in the system to be built.

We will call the complete model of users, user roles, and use cases, the usage model. The usage model is typically supplemented by a more conventional domain model that represents entities or constructs from the problem domain and their interrelationships. This serves to define the "domain of discourse" between user/client and developer. The usage model is expressed in terms of the domain model. The domain model itself may take the form of an entity-relationship diagram or an object-oriented extension, such as in the Extended Uniform Object Notation (Henderson-Sellers, Edwards, and Constantine, 1992).

Entity, Interface, and Control Objects

If our systems are to have robust architecture that responds readily to new functional needs in terms of expanded or changing users, user roles, and usage, the architecture of the software should reflect the essential structure of our usage model. Unfortunately, many object-oriented methods use an entity-oriented or application domain model as the primary or exclusive basis for the design. These push the designer into allocating all behavior to objects derived from the domain model. More robust designs result from differentiating three types of object modules each with a different purpose or function in the system.

Entity objects model information in the system that must be retained over time. Usually this means information that survives from one use case to another. All behavior naturally and closely related to this information is placed within the entity object. Entity objects nearly always correspond to entities in an application domain model, that is, to tangible objects or conceptual entities within the application area. An example of an entity object in a personnel management system might be an employee with its associated data defining name, skills, employment history, etc.

Interface objects mediate between actors and the rest of the software system. They encapsulate functionality specific to particular devices or actors, thus allowing for changing interface characteristics independently of underlying functional capability and vice versa. Interface objects model behavior and information that is dependent on the interface of the system, thus, everything specifically about any interface of the system is placed in an interface object.

Control objects model behavior that is not naturally tied to any other object. Typically this behavior involves interaction with several different entities. They encapsulate policies or procedures that span multiple object classes or complex behaviors involving various classes. (Page-Jones and Weiss in their "Synthesis" method refer to these as event managers. See Page-Jones and Weiss, 1989; Weiss and Page-Jones, 1992.) An example might be a control object to encapsulate the lending policy of a college library, where this policy involves the kind of borrower, the type of material, and even the time in the semester. Another example would be the polling object in a patient monitoring system that polls and reports on various patients in different wards according to a priority scheme. In both of these examples, the behavior could be allocated to other objects (such as the borrower or the patient), but the resulting design is clumsier and less robust because the behavior does not really belong to those other specific objects.

Typically, a control object will correspond to a use case. In the actual operation of the software, a control object is instantiated for each instance of the particular use case and remains for the duration of the interactive sequence.
In developing object-oriented models, interface objects and control objects are distinguished from entity objects by small annotations. These are shown in Figure 1 in the form used with the Uniform Object Notation (Page-Jones, Constantine, and Weiss, 1990; Henderson-Sellers, Edwards, and Constantine, 1992). Thus the essential purpose of object modules in the design is evident from the model.

The most stable systems cannot be built using only objects that correspond to “real-world” or problem domain entities. For the most robust design, the effects of changes must be localized so that they affect only one object. Distributing behavior systematically over entity, interface, and control objects helps to isolate the effects of changes. Changes to an interface should typically affect only interface objects. A change in behavior associated with information held by the system will typically affect only the entity object representing that information. And changes in functionality spanning multiple objects are likely to be found isolated into a control object within an appropriate design.

The Usage Model

The usage model includes a list and description of all users, a list and description of all actors (user roles), and a model of all use cases. Each use case is named and the entire sequence of stimuli, actions, and responses is described.

Use cases are typically interrelated. Use-case driven design recognizes interrelationships between use cases that can simplify the use case model. One use case may be a specific example of or subset of another, for example, an “account balance request” is a kind of “query.” This relationship, inherits or “is-a” can be used to model a collection of complex interrelated use cases more simply. For example, in an ATM application, “withdrawal” and “deposit” inherit “customer transaction” because all instances of “withdrawal” and “deposit” are examples of “customer transaction.”

Another relationship is extension. A use case can be defined as extending the behavior of another. If use case A “extends” use case B, then the full functional behavior of A includes all of that of B with new behavior defined in A inserted. For example, in an ATM application, the use case “customer ATM card gets stuck” may include efforts to dislodge the card, displaying a screen message and sending an alarm condition to a central office. But this use case “extends” the normal customer transaction of deposit, withdrawal, or query. The complete sequence of actions and responses associated with “customer ATM card gets stuck” takes place within the sequence of a deposit, withdrawal, or query. Like a subroutine in a functional hierarchy, it need only be described once rather than as a part of all the other use cases it extends.

Strategy

The purpose of this discussion has been to introduce the notion of a use case and the distinctions among interface, entity, and control objects. It is beyond the scope of this brief introduction to cover the entire procedure of use case driven design. (See Jacobson et al., 1992.) In somewhat simplified outline, the elements special to the strategy of developing a use case driven design for an embedded systems application are as follows:

1. Identify all users interacting with the system (persons, devices, other systems).
2. Identify all the actors (user roles).
3. Identify the use cases (interaction scenarios) associated with each actor.
4. Model the behavior (actions, responses) for each use case.
5. Model the conceptual structure of entities in the application domain to form a domain model.
6. Allocate behavior from use cases to entity objects, interface objects, and control objects.

In allocating behavior from the use cases to object modules, certain general rules are followed. Broadly speaking, the goal is to keep closely related behavior together and less related behavior apart. The initial preference is to allocate behaviors to entity or interface objects. As a rule, an entity object should encapsulate only those operations directly involved with setting, accessing, altering, or otherwise operating upon the encapsulated data structures that define the attributes of that one entity. Behaviors that do not have to do with interfaces, that are not easily assigned to a particular entity object, or that involve multiple objects (interface or entity objects) are candidates for placement in control objects. Control objects are a powerful architectural tool, but it is important not to use them excessively or the result can be a function-oriented design cast in the form of objects. Such “corrupted” designs are almost always less robust.

REFERENCES

In developing object-oriented models, interface objects and control objects are distinguished from entity objects by small annotations. These are shown in Figure 1 in the form used with the Uniform Object Notation (Page-Jones, Constantine, and Weiss, 1990; Henderson-Sellers, Edwards, and Constantine, 1992). Thus the essential purpose of object modules in the design is evident from the model.

The most stable systems cannot be built using only objects that correspond to "real-world" or problem domain entities. For the most robust design, the effects of changes must be localized so that they affect only one object. Distributing behavior systematically over entity, interface, and control objects helps to isolate the effects of changes. Changes to an interface should typically affect only interface objects. A change in behavior associated with information held by the system will typically affect only the entity object representing that information. And changes in functionality spanning multiple objects are likely to be found isolated into a control object within an appropriate design.

The Usage Model

The usage model includes a list and description of all users, a list and description of all actors (user roles), and a model of all use cases. Each use case is named and the entire sequence of stimuli, actions, and responses is described.

Use cases are typically interrelated. Use-case driven design recognizes interrelationships between use cases that can simplify the use case model. One use case may be a specific example of or subset of another, for example, an "account balance request" is a kind of "query." This relationship, inherits or "is-a" can be used to model a collection of complex interrelated use cases more simply. For example, in an ATM application, "withdrawal" and "deposit" inherit "customer transaction" because all instances of "withdrawal" and "deposit" are examples of "customer transaction."

Another relationship is extension. A use case can be defined as extending the behavior of another. If use case A "extends" use case B, then the full functional behavior of A includes all of that of B with new behavior defined in A inserted. For example, in an ATM application, the use case "customer ATM card gets stuck" may include attempts to dislodge the card, displaying a screen message and sending an alarm condition to a central office. But this use case "extends" the normal customer transaction of deposit, withdrawal, or query. Like a subroutine in a functional hierarchy, it need only be described once rather than as a part of all the other use cases it extends.

Strategy

The purpose of this discussion has been to introduce the notion of a use case and the distinctions among interface, entity, and control objects. It is beyond the scope of this brief introduction to cover the entire procedure of use case driven design. (See Jacobson et al., 1992.) In somewhat simplified outline, the elements special to the strategy of developing a use case driven design for an embedded systems application are as follows:

1. Identify all users interacting with the system (persons, devices, other systems).
2. Identify all the actors (user roles).
3. Identify the use cases (interaction scenarios) associated with each actor.
4. Model the behavior (actions, responses) for each use case.
5. Model the conceptual structure of entities in the application domain to form a domain model.
6. Allocate behavior from use cases to entity objects, interface objects, and control objects.

In allocating behavior from the use cases to object modules, certain general rules are followed. Broadly speaking, the goal is to keep closely related behavior together and less related behavior apart. The initial preference is to allocate behaviors to entity or interface objects. As a rule, an entity object should encapsulate only those operations directly involved with setting, accessing, altering, or otherwise operating upon the encapsulated data structures that define the attributes of that one entity. Behaviors that do not have to do with interfaces, that are not easily assigned to a particular entity object, or that involve multiple objects (interface or entity objects) are candidates for placement in control objects. Control objects are a powerful architectural tool, but it is important not to use them excessively or the result can be a function-oriented design cast in the form of objects. Such "corrupted" designs are almost always less robust.

REFERENCES

Implementing Syntax-Driven Object-Oriented Design

Jack Crenshaw
Tampa, Fla.

Jack Crenshaw is an independent consultant and a contributing editor to Embedded Systems Programming. He wrote his first computer program in 1954, and has been involved with software design, development, and methodologies ever since. He did much early work in the space program and has developed many analysts and real-time programs. He is the author of the Programmer's Toolbox column in Embedded Systems Programming. Crenshaw holds a Ph.D. in Physics from Auburn University. Computer Science he learned the hard way.

No paper available.
Contact author for further information.