Function-Class Decomposition: A Hybrid Software Engineering Method

Function-class decomposition—a simple yet powerful hybrid method that integrates structured analysis with an object-oriented approach—provides a supportive architecture for modeling large, complex software systems into a hierarchy of functional modules.

Carl K. Chang
Jane Cleland-Huang
University of Illinois, Chicago

Shiyan Hua
Lucent Technologies

Annie Kuntzmann-Combelles
Q-Labs

As software systems have grown consistently larger and their functionality has become increasingly complex, the advantages of using object-oriented methods to improve their organization and structure have become apparent. OO methods use packages, components, and subsystems to provide a systematic approach to software development and maintenance, but they do not provide strong guidelines for using these constructs to partition complex systems. In addition to offering a simple yet powerful method for decomposing a system, function-class decomposition[1] produces an architecture that is more supportive than traditional OO decomposition for several software engineering tasks.

FCD, a hybrid method that integrates structured analysis with an OO approach, identifies classes in parallel with decomposing the system into a hierarchy of functional modules. Recently, FCD was extended to integrate UML concepts.[2] Useful for partitioning a system for distribution,[3] the FCD hierarchy provides a framework for controlling development in a distributed software engineering environment. It also helps to identify and integrate components in component-based development and supports the system life-cycle maintenance phase. In addition, FCD addresses many of the initial analysis and design problems inherent in large and complex OO systems.

We have tested the FCD approach on several applications, including UICCELL, a large-scale mobile phone simulator; M-Net, an Internet-based conferencing system; and a mobile agent-based system for collaborative requirements engineering built on top of an industrial-strength requirements management system.

NEED FOR STRUCTURE

Although some leading OO researchers initially declared orthogonal approaches such as structured analysis and OO mutually exclusive,[4] practitioners have become increasingly aware that large, complex applications need structure. The use of subsystems and components can help reduce a large system's complexity and provide an initial decomposition of it. Using a layered architecture can create a high-level system decomposition, but it partitions the system into vertical layers instead of component-like functional modules, and it lacks many of the benefits that FCD provides. In an extensive survey of OO methods, Roel Wieringa[5] concluded that they do not provide adequate guidelines for partitioning a system.

The developers of the Mars Pathfinder Mission give a telling account of their experience with the project's flight software architecture.[6] As instructed by textbook OO methodologies, they began by enumerating as many objects as they could identify, intending to place them into a flat object space. The team then planned to arrange the objects into subsystems by identifying groups of closely collaborating objects. But they immediately encountered problems because a superficial effort identified 40 objects, and that number expanded rapidly as they explored mission requirements. The team concluded that this bottom-up approach was impractical in relation to their project's size and com-

0018-9162/01/$17.00 © 2001 IEEE

FCD can help software engineers experienced in structured analysis make the transition to OO development.

plexity. Instead, they chose an approach that combined top-down functional decomposition's ability to compartmentalize the problem with OO decomposition's ability to identify and model the behavior within each compartment.

STRUCTURED VERSUS OBJECT-ORIENTED APPROACH

Structured analysis evolved in the 1970s and 1980s in response to the need for a decomposition technique for analyzing complex software applications. Early structured methodologies all endorsed functional decomposition. For example, Ed Yourdon's[7] top-down decomposition technique used dataflow diagrams to depict processes and the data flow between them. Still in use today, these dataflow diagrams serve as the basis for *transformation*, a systematic process for deriving structure charts to provide a top-level design view.

Using a structured approach requires observing key principles such as the importance of information hiding. Failure to do so can result in difficult-to-maintain systems that lack extensibility and cannot be reused in other contexts. The OO paradigm addresses this issue by encapsulating data and its corresponding operations within a class. The identification of entities and their interrelationships provides the basis for this approach. Class-oriented decomposition identifies real-world entities, models them as high-level classes, and decomposes them into lower-level classes. Performing several iterations of the class and relationship identification process refines the identified classes to derive the software system's architecture. The OO approach and its related methodologies have enjoyed wide acceptance, and this approach has replaced the older structured methods in many software projects.

Using a UML package is a popular method of organizing an OO system. These packages serve as containers for bundling together classes, hiding their internal structure to provide a more abstract view of the system. UML reference materials[2] describe how to use a set of existing interrelated classes to identify packages. However, this process suggests a bottom-up rather than top-down approach to packaging, which fails to address the problem of applying structure to the decomposition process.

Using subsystems in decomposition

James Rumbaugh[8] proposed organizing classes into subsystems that provide both high-level abstract views and low-level implementation views of a system. In Rumbaugh's approach, the developer first builds a high-level model of the system's structure and then decomposes the model into subsystems. The decomposition process builds a hierarchical structure iteratively, constructing subsystems for each subsystem, which can in turn contain other subsystems. The underlying subsystem structure supports both a high-level abstract view and a lower-level detailed view.

Identifying subsystems presents a challenge when building a system this way. Early in the decomposition process, developers cannot always determine whether an identified class should form a subsystem itself or become a member of one. Maintaining integrity between layers when making changes also poses problems.

FCD methodology is especially useful in decomposing medium- to large-sized systems. It can also help software engineers experienced in structured analysis make the transition to OO development. FCD maintains integrity between layers by using a simplified notation at the higher layers that facilitates upward synchronization. Developers find FCD's simple notation easy to learn, expressive, and suitable for the high-level and often fuzzy thought processes that occur at the start of the decomposition process.

Integrating structured and object-oriented analysis

Researchers periodically visit the idea of integrating structured and OO analysis. Larry L. Constantine[9] used structured analysis to analyze and specify the internal function of classes, but this approach does not provide structure for a complex system. He suggested but did not elaborate upon the idea of imposing a hierarchy of control on the classes in a system—the type of integration that FCD uses.

In response to the lack of methods for managing entities in traditional structured analysis, Sidney Bailin[10] developed a parallel process of decomposing objects and allocating functions. He also introduced entity dataflow diagrams, which contain entity and function nodes, and the related process of decomposing entities into subentities and functions into subfunctions. Bailin's method requires every function to occur within the context of an entity. In contrast, FCD decomposes functionality in parallel with class identification. Another major difference is that Bailin's method requires engineers to perform entity-relationship modeling, a skill that some software engineers, such as those working in real-time embedded systems or reactive and control systems, may not possess. Further, his method produces an intertwined presentation of the entity and function decomposition that requires mentally separating the two types of processes to understand the model. FCD facilitates reading the model by using a simpler notation that clearly delineates the functional modules and classes.

Wieringa[5] argued that because structured and OO methods actually use similar decomposition criteria, the distinction between the two is artificial. He pointed

Function-Class Decomposition Algorithm

When applying function-class decomposition, use the top-down approach described in the following steps:

1. Treat the entire system initially as a single high-level functional module at level 1. Specify high-level requirements and their related scenarios.
2. Use requirements and scenarios to identify an initial set of classes that fulfill the basic functionality of the level 1 functional module. Use FCD notation to represent these classes within the functional model.
3. Within each functional model, observe responsibilities and collaborations of each class and place classes into the subgroups that appear to maximize internal cohesion and minimize external coupling.
4. Develop use-case maps for each functional model. Select key scenarios and analyze their paths to assess each subgroup's cohesion and coupling. Rearrange classes within the subgroups to minimize external coupling and maximize internal cohesion, if necessary.
5. For each group of classes, form a functional model at the next FCD hierarchy level and name it meaningfully. Each new functional model will initially contain only the classes grouped together at the previous level.
6. Allocate to each new functional model the requirements and scenarios it can completely fulfill. Other requirements and scenarios that span functional models at the new level should remain attached to the parent functional model.
7. For each functional model at the new level, analyze and refine the requirements and scenarios allocated to the module, using them to identify additional classes.
8. Repeat steps 3 through 7 until the system requires no further decomposition, usually after all its major functionality has been identified, but sometimes earlier when using FCD to partition a system for work allocation in a distributed development environment.
9. If all major system functionality has been identified, or if decomposing the remainder of the system using a more traditional OO approach, generate UML class diagrams for each leaf node. Once the class diagrams are generated, additional support classes may be added.
10. Starting at the lowest level of the hierarchy, use scenarios allocated to the functional model to define functional model interfaces. Move up the hierarchy systematically, considering only the intermediate-level functional models with child modules that have defined class diagrams. Continue the process until it reaches the top-level functional model and the entire system has been integrated.

out that incompatibility occurs only in dataflow diagrams that represent the sequence of functional transformations needed to convert system inputs to outputs and activity charts that separate data, operations, and control entities. He emphasized that other structured techniques such as functional decomposition, event partitioning, device-oriented decomposition, and subject-domain-oriented decomposition are in fact compatible.

UNDERSTANDING FCD

FCD, an iterative process, applies a top-down approach to decomposing a system while simultaneously identifying and grouping classes into functional modules, with each module representing a specific functionality the system requires. The "Function-Class Decomposition Algorithm" sidebar summarizes this iterative process.

A module acts as the container for a group of classes that exhibit high internal cohesion and low external coupling. As Figure 1 shows, the main decomposition tasks consist of identifying classes, placing classes into groups, using the class groupings to form functional modules at the next level, and allocating scenarios to appropriate functional modules. System integration then occurs from the bottom up.

Although we can apply FCD as one stage in a waterfall model, it fits best into an iterative model. In fact, FCD was initially developed as part of the software-architecture-based requirements engineering environment (SABRE). The interdependency of requirements and the software architecture are the pivotal factors in the SABRE development process. The requirements drive and validate the software architecture's incremental development. The iterative FCD process elicits and validates requirements at several levels. FCD notation consists of typical UML notations for classes, with rectangular bars representing the functional modules.

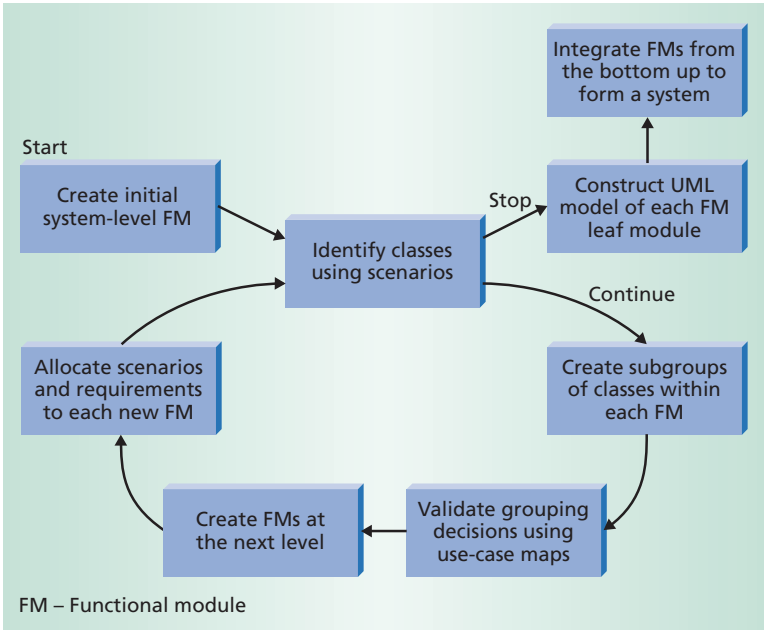

Figure 1. Function-class decomposition process. An iterative process elicits and validates requirements at several levels, followed by bottom-up system integration.

Class identification

FCD treats the entire system as a single abstract module representing the system's high-level function-

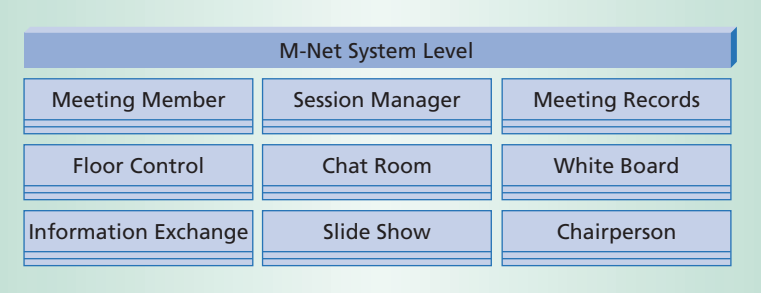

Figure 2. M-Net system-level functional module and related classes. FCD initially identified nine classes in the system-level module.

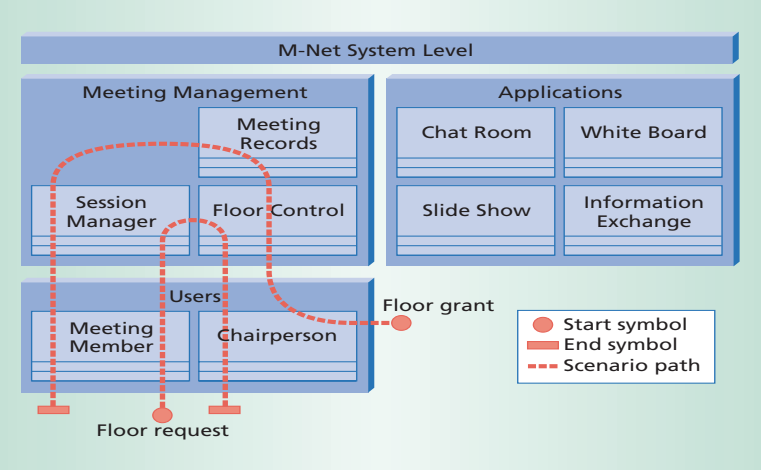

Figure 3. Initial grouping decisions and scenario mappings for an M-Net system-level module using a use-case map. The curved line represents a scenario's path as it weaves its way through the components with which it interacts.

ality. Scenarios refine system-level requirements and identify high-level classes within the functional module. The decomposition's starting point consists of a single functional module and the set of classes that represent the module's behavior. Figure 2 shows a system-level module from the decomposition of M-Net, a real-time Internet-based collaborative meeting application.[11] As this example shows, we initially identified nine classes in the M-Net system-level module.

Class grouping

FCD identifies the high-level classes and divides them into subgroups. A subgroup should exhibit relatively strong internal cohesion between the classes in the group and weak external coupling with classes in other subgroups. In our experience, the optimum group size is five to nine classes, but a group could consist of just one class if that class could be decomposed into several more refined classes. Successful decomposition depends on identifying appropriate class groupings.

Initial attempts to use FCD to group classes by analyzing their interconnections[1] worked poorly because practitioners usually are unaware of specific details about the interaction between classes at early stages of decomposition. In addition, a static analysis of the class structure does not provide all the information needed to determine the level of coupling between classes. For example, the presence of a bidirectional link between classes could be a sign of close coupling or could result from a callback reference at work, which indicates looser and more desirable coupling between the classes. Similarly, the presence of a specialization relationship does not necessarily indicate close coupling, as in the Java class library in which all classes derive from the object class. Because of the difficulties associated with grouping classes by examining their static relationships, we use a more dynamic approach based on how scenarios interact with classes and subgroups.

FCD examines the responsibilities and collaborations of classes in the functional module and places the classes into groups that appear to maximize internal cohesion and minimize external coupling. It places extra classes that do not obviously fit into any specific group into a miscellaneous or library group. Use-case maps provide a visual notation for mapping scenarios onto components and help determine if the groupings meet cohesion and coupling criteria.[12]

Ray Buhr[12] observed that cohesion and coupling could be measured by analyzing scenario mappings onto system components. They found high cohesion when all the scenarios that interact within a component have related functionality and low cohesion when scenarios with different functionality interact extensively. Observing the paths that scenarios take can also measure coupling. High coupling occurs when a scenario path weaves its way excessively between two components. Examining the path each scenario takes tests each subgroup for cohesion and coupling. After iteratively rearranging class groupings to achieve satisfactory results, each subgroup receives a meaningful name that represents its functionality.

Grouping decisions

Figure 3 shows the grouping decisions made for level 1 of the M-Net example. At this high level, FCD groups classes according to the high-level functionality of meeting management, applications, and users. During an M-Net meeting, when a member requests the floor, the following sequence of events occurs:

- The member requests the floor by sending a message through the session manager and floor controller to the chairperson.
- The chairperson updates the floor request list to reflect the new request.
- To grant the floor to the requesting member, the chairperson selects that member from the floor request list and issues a grant command.
- The system updates floor control to reflect that it will grant the floor to the requesting member.
- The floor controller sends a message, via the ses-

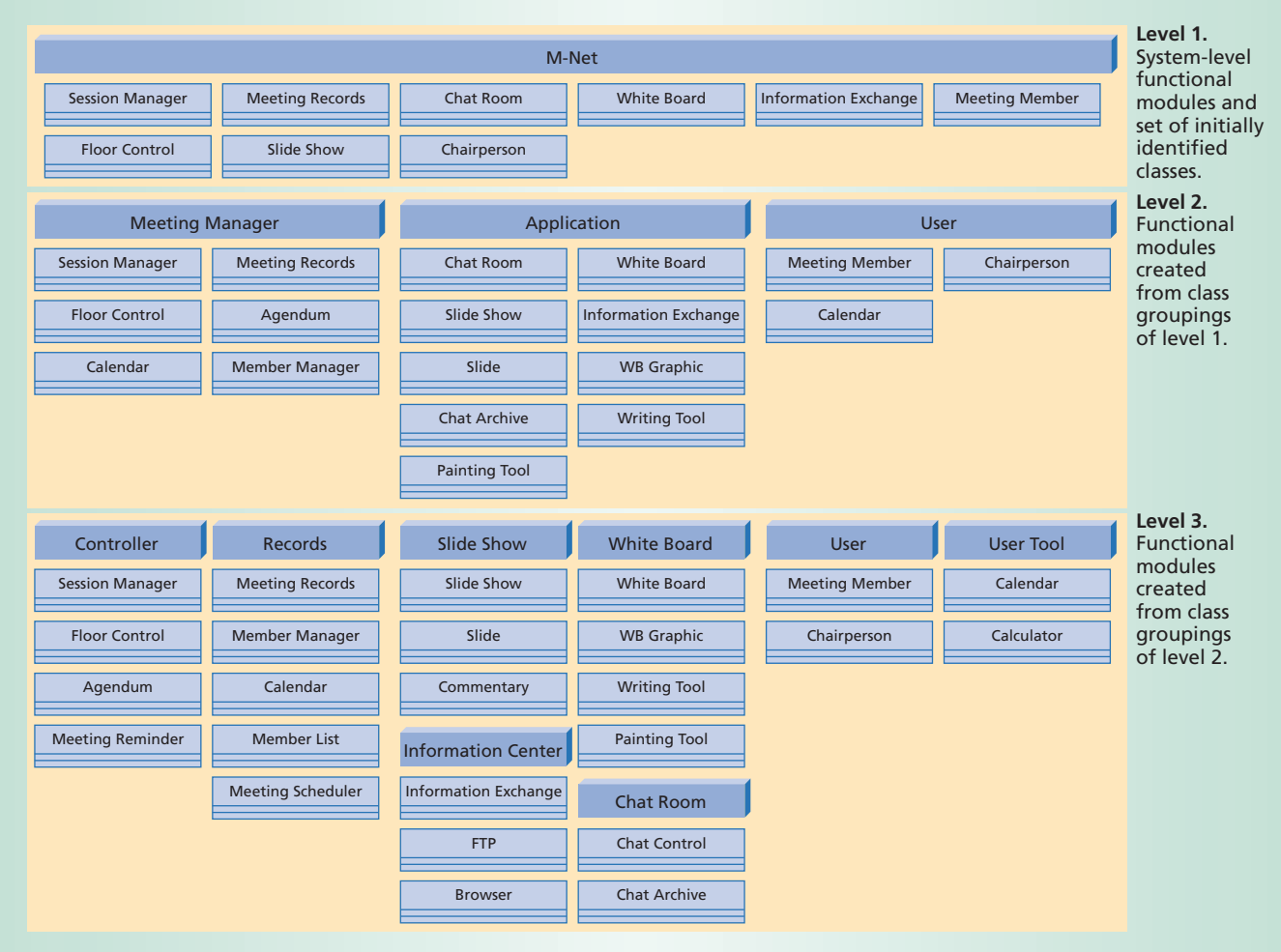

Figure 4. Function-class decomposition of M-Net to three levels. The identified and validated class groupings from the previous level form functional modules at the next level.

sion manager, to the requesting member indicating that he has the floor.

The scenarios do not indicate any undesired levels of cohesion or coupling; a more complete mapping of scenarios to components would further validate the grouping decisions.

Allocation of requirements and scenarios

To construct the hierarchy, FCD uses the validated class-groupings from the previous level to form functional modules at the next level. Figure 4 shows decomposition of the M-Net example into three levels. Unsurprisingly, the groupings often indicate more than one possible solution to the problem. For example, in Figure 4, FCD could place the chat room, white board, slide show, and information exchange applications from level 1 into their own functional modules at level 2.

FCD analyzes each scenario path to determine whether its behavior is encapsulated within one new functional module or whether it spans several new modules. If a new module encapsulates the behavior, FCD allocates the scenario and related requirements to that module. Otherwise, the scenario remains allocated to the functional module at the current level. This process facilitates the refinement and further decomposition of the system within each functional module and maps scenarios and requirements to specific levels of the architecture, which helps determine the impact of change during the software maintenance stage.

Refining the decomposition

Once FCD creates functional modules at the next level, the process begins again. It uses a more refined set of requirements and scenarios to identify more classes within each functional module. During this stage of class identification, the developer refines existing classes, adds additional classes that represent new functionality, and considers rearranging classes to follow proven design patterns for solving specific types of problems. Next, the developer analyzes the extended set of classes for further grouping opportunities. Because decomposition refines scenarios as it progresses, scenarios at lower levels can decompose functional modules and classes to a finer granularity.

Two techniques can determine how far FCD should go. One approach continues decomposing each functional module until it is unlikely that additional classes will be identified for that module. A second approach determines the level in the decomposition process at which more traditional OO modeling is appropriate for mapping the FCD functional modules.

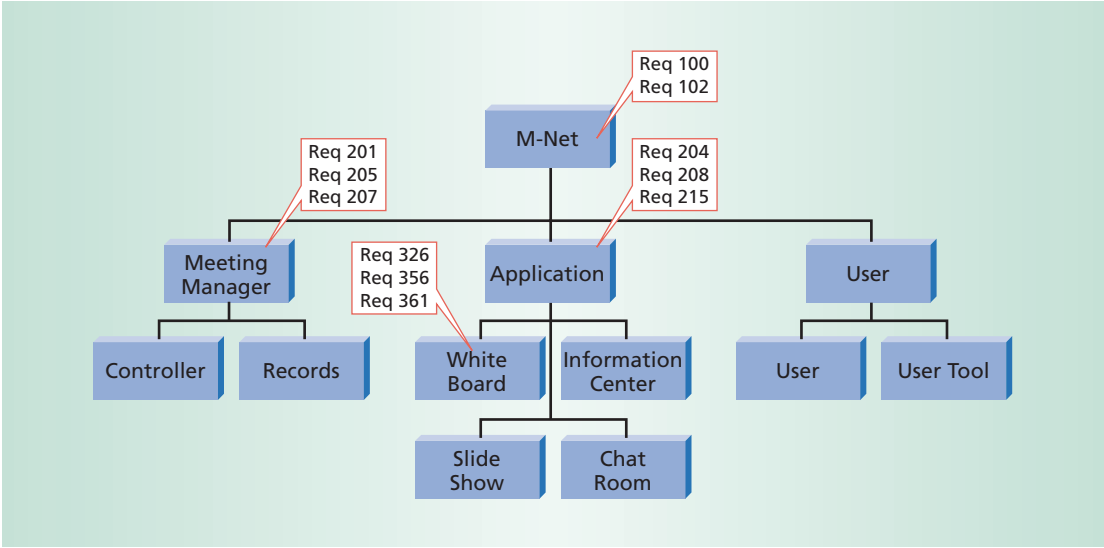

Figure 5. Functional view showing a selection of requirements allocated to specific modules. The leaf modules represent low-level black-box subsystems. FCD can recursively combine more primitive functional modules to construct intermediate subsystems.

When using the second approach, FCD primarily identifies upper- and middle-level subsystems. When developing an application in a distributed software engineering environment, we can use this approach to perform an initial FCD that partitions the work into distributable units for further decomposition using FCD or another approach. Low-level system decomposition may require using class diagrams and other models to represent highly complex interrelationships between classes.

System integration

Once FCD completes top-down system decomposition, it performs bottom-up integration and interface definition. It generates a UML class diagram for each of the leaf nodes, using the attached scenarios to identify specific class interactions. Each leaf node is a primitive component or subsystem. Moving iteratively up the hierarchy, FCD selects an intermediate node for which it has already defined all its children as class diagrams. The scenarios attached to this node represent behavior that spans at least two of the child functional modules. FCD uses these scenarios to refine individual class diagrams and to establish external interfaces between them. This process continues until FCD reaches the top-level functional module and finishes integrating the entire system. During the integration process, FCD can add classes to each functional module that facilitate integration.

When FCD uses a component-based integration framework, it treats the functional modules as components and places them into relevant parts of the framework. The components can either be primitive functional modules taken from the leaves of the FCD hierarchy or composite components constructed from subtrees of the hierarchy.

Supported views

FCD supports three types of system views. The FCD view, as illustrated in Figure 4, shows both the functional modules and the classes they contain at each of the three levels. This view reflects the decomposition process itself, as it records the process of identifying and grouping classes and forming functional modules. FCD can attach rationale documents containing textual descriptions and links to requirements to each hierarchy level. A simple traceability scheme can augment the FCD view. Explicit traceability preserves certain refinements such as the relationship between a class at a higher level and the multiple classes decomposed into a lower level. In this sense, FCD supports not only the decomposition process but also the equally important job of preserving and documenting the process.

To generate a hierarchical view that shows the application's structural model, FCD hides all the classes and shows only the functional modules. FCD can recursively combine more primitive functional modules to construct intermediate subsystems. The third view resembles a more traditional OO class diagram. FCD constructs it from the bottom up, using scenarios allocated to each functional module to define class diagrams and external interfaces. It then uses these interfaces to generate a systemwide class diagram.

FCD STRENGTHS AND WEAKNESS

FCD requires extra work and discipline during the decomposition process to identify and group classes into functional modules. However, in our experience, the benefits justify this additional effort. The FCD hierarchy supports a number of software engineering lifecycle tasks. As system requirements change over time, engineers can use FCD's hierarchy mapping requirements to determine the scope of the changes. Requirements mapped to leaf nodes affect only the primitive components those nodes represent, whereas requirements attached to higher-level functional modules affect multiple system components. For example, in Figure 5, if a change request specifies a modification to requirement 326, it is immediately apparent that the scope of the change would be limited to the white board module. In contrast, a change request against requirement 102 could have a systemwide effect.

FCD's hierarchical structure offers the distinct advantage that detailed changes made at lower levels—for example, the decision to use aggregation instead of inheritance—do not require adjustments at higher levels because that type of detailed information is not part of the higher-level model. Only changes that require reassignment of classes to a different functional module defined at a higher level in the hierarchy require higher-level adjustments.

FCD's hierarchy also offers the advantage of test-case-generation support. It attaches scenarios to leaf nodes to generate unit tests, and scenarios attached to higher-level nodes generate integration and system-level tests. Finally, the natural boundaries between functional modules in the FCD hierarchy simplify the task of identifying initial partition points in distributed systems.[3]

Our experience validates FCD's compatibility with OO methodologies and modeling techniques. FCD uses an iterative, top-down process of subsystem identification and decomposition that reduces the inherent complexity of more traditional OO modeling approaches. In addition to supporting the decomposition process itself, the resulting FCD architecture and related artifacts support maintenance of the system in the face of changing requirements. Future work will focus on extending our knowledge of FCD by investigating its application to a much broader range of systems and refining the approach to support other aspects of a typical software life cycle. ✸

References

1. C.K. Chang and S. Hua, "A New Approach to Module-Oriented Design of OO Software," *Proc. 18th Int'l Computer Software and Applications Conf.* (COMPSAC 94), IEEE CS Press, Los Alamitos, Calif., 1994, pp. 29-34.
2. G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Addison Wesley Longman, Reading, Mass., 1999.
3. J.L. Huang and C.K. Chang, "Supporting the Partitioning of Distributed Systems with Function-Class Decomposition," *Proc. 24th Int'l Computer Software and Applications Conf.* (COMPSAC 00), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 351-356.
4. G. Booch, *Object-Oriented Analysis and Design with Applications*, Benjamin/Cummings, San Francisco, 1994.
5. R. Wieringa, "A Survey of Structured and Object-Oriented Specification Methods and Techniques," *ACM Computing Surveys*, Dec. 1998, pp. 459-527.
6. S. Stolper, "Streamlined Design Approach Lands Mars Pathfinder," *IEEE Software*, Sept./Oct. 1999, pp. 52-62.
7. E. Yourdon, *Modern Structured Analysis*, Yourdon Press, Upper Saddle River, N.J., 1989.
8. J. Rumbaugh, "Building Boxes: Subsystems," *J. Object-Oriented Programming*, Oct. 1994, pp. 16-21.
9. L. Constantine, "Objects, Functions, and Program Extensibility," *Computer Language*, Jan. 1990, pp. 74-82.
10. S. Bailin, "An Object-Oriented Requirements Specification Method," *Comm. ACM*, May 1989, pp. 608-623.
11. C.K. Chang et al., "Rule-Mitigated Collaboration Technology," *Proc. 24th Workshop on Future Trends of Distributed Computing Systems* (FTDCS 99), IEEE CS Press, Los Alamitos, Calif., 1999, pp. 137-142.
12. R.J.A. Buhr, "Use-Case Maps as Architectural Entities for Complex Systems," *IEEE Trans. Software Engineering*, Dec. 1998, pp. 1131-1151.

Carl K. Chang *is the director of graduate studies in computer science in the Department of Electrical Engineering and Computer Science at the University of Illinois, Chicago. His research interests include software engineering and net-centric computing. Chang received a PhD in computer science from Northwestern University. He is an IEEE Fellow and a member of the IEEE Computer Society. Contact him at chang@uic.edu.*

Shiyan Hua *is a member of the technical staff in the Intelligent Network Messaging System Wireless Service Department at Lucent Technologies. She received a PhD in electrical engineering and computer science from the University of Illinois, Chicago. She is currently working on the Universal Mobile Telecommunications System defined by the European Third Generation Partnership Project. Contact her at shua@lucent.com.*

Jane Cleland-Huang *is a doctoral candidate in the Department of Computer Science at the University of Illinois, Chicago. Her research interests include requirements engineering, object-oriented modeling, and requirements-based traceability frameworks. She is a member of the IEEE and the ACM. Contact her at jhuang1@cs.uic.edu.*

Annie Kuntzmann-Combelles *is an executive vice president at Q-Labs. Her research interests include process engineering, risk management, and object-oriented requirements and design. She is a graduate of the Ecole Nationale Supérieure de l'Aéronautique et de l'Espace. Contact her at akc@objectif.fr.*