User Interface Modeling in UML*i*

Paulo Pinheiro da Silva, *Stanford University*

Norman W. Paton, *University of Manchester*

Although user interfaces represent an essential part of software systems, the Unified Modeling Language seems to have been developed with little specific attention given to user interface issues. You can use standard UML to model important aspects of user interfaces,[1] but this often results in unwieldy and unnatural representations.[2–4]

Modeling user interfaces is a well-established discipline in its own right. For example, modeling techniques can describe interaction objects,[5] tasks,[6] and lower-level dialogs in user interfaces.[7] For more than 10 years, researchers have tried to build comprehensive interface development environments that integrate models for describing a user interface's different aspects. Many research projects have addressed model-based user interface development environments (MB-UIDEs)[8] to provide specialized models for describing user interfaces. These models often include facilities that produce interfaces that can run from declarative descriptions of their behavior.[9] Using models as part of user interface development can help capture user requirements, avoid premature commitment to specific layouts and widgets, and make the relationships between an interface's different parts and their roles explicit.

Although MB-UIDEs provide models that effectively capture user interface functionality, most proposals provide limited facilities for modeling user interfaces along with an application's other aspects. For example, most MB-UIDEs have a domain model that describes the data over which the interface acts but provide limited facilities for describing the functionality of the application for which the interface is being constructed. Thus, MB-UIDEs are weak in application modeling, an area of specialization for UML. By using a modeling environment in which application and interface designers describe models in terms of an integrated set of

Object modeling languages rarely address user interface issues. However, UML*i* conservatively extends UML with explicit support for interface modeling.

notations, you can enhance communication between design team members.

Integrating application and interface modeling facilities

Integrating UML's application modeling facilities with MB-UIDE's interface modeling facilities might offer mutual benefits. Clearly, grafting notations from one community into the other's context could easily lead to complex and inelegant proposals in which independently developed models share overlapping capabilities and present users with challenges such as when and how to use different notations.

The following principles can help integrate user interface modeling facilities into UML:

- The integrated proposal should be unobtrusive, retaining standard UML as a subset in which existing constructs keep their roles and semantics.
- The integrated proposal should support the expectations of current UML modelers, whose experience with UML should help when using interface-specific extensions.
- The integrated proposal should support the expectations of user interface modelers who have experience using existing interface-modeling techniques. Such users should not feel they must design interfaces with more limited facilities than the MB-UIDEs provide.
- The integrated proposal should support complete applications, so links between user interface models and existing UML models should be well-defined and close.
- The integrated proposal should introduce as few new models into UML as possible.

Several researchers have investigated integrating interface modeling techniques with UML. For example, one approach assesses UML models for use in interface modeling, comparing them with a collection of specialist interface modeling notations.[2] Another approach suggests how you can use several UML models—particularly, class diagrams and use-case diagrams—along with task models for user interface modeling.[4] Another approach lets you comprehensively model Web applications,[10] but this approach is less conceptual than the other proposals.

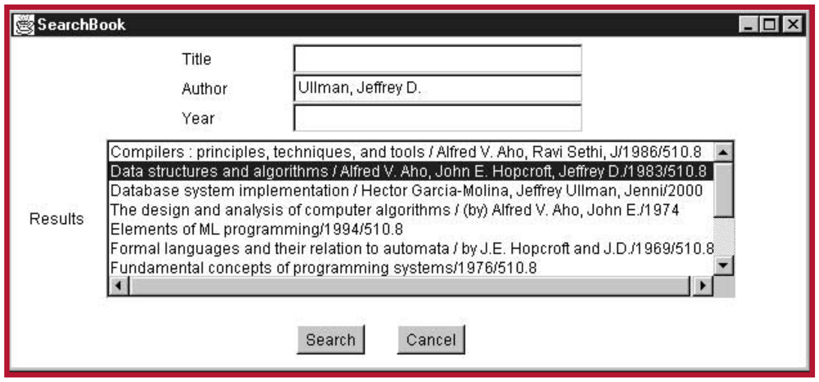
In UML*i*, you model tasks using extended activity diagrams rather than by incorporating a completely new notation into UML. UML*i* also addresses the relationships between use cases, tasks, and views, and thoroughly addresses the relationship between tasks and the data on which they act. UML*i* is probably the most technically mature proposal for interface development in UML. The UML*i* metamodel fully integrates with UML. And you can build and integrate UML*i* models with other UML models in an extension to the ArgoUML toolset.[11] You can download the UML*i* extension to ArgoUML from http://img.cs.man.ac.uk/umli.

Overall, although the relationship between UML and user interface modeling languages has sparked growing interest, the field is open to new proposals and ideas, especially because no widely accepted solution for user interface modeling in UML exists. This article examines some of UML's user interface modeling facilities. Figure 1 shows the `SearchBook` interface, an example of the sort of interface our method supports. In this example, you can provide a combination of a book's properties for querying a library database.

You can press the search and cancel buttons in the `SearchBook` interface at any time when the interface is visible. You submit a query by pressing the search button. The library application uses the parameters available in the interface when you press the search button to build the query. The application then displays the results for the last query submitted within the current session in the results list. Pressing the cancel button quits the `SearchBook` interface and returns you to the main interface.

Modeling user interfaces

For many years, researchers have worked on



Author: Ullman, Jeffrey D.

Compilers : principles, techniques, and tools / Alfred V. Aho, Ravi Sethi, J/1986/510.8
Data structures and algorithms / Alfred V. Aho, John E. Hopcroft, Jeffrey D./1983/510.8
Database system implementation / Hector Garcia-Molina, Jeffrey Ullman, Jenni/2000
The design and analysis of computer algorithms / (by) Alfred V. Aho, John E./1974
Elements of ML programming/1994/510.8
Formal languages and their relation to automata / by J.E. Hopcroft and J.D./1969/510.8
Fundamental concepts of programming systems/1976/510.8

Figure 1. A screen shot of the `SearchBook` user interface.

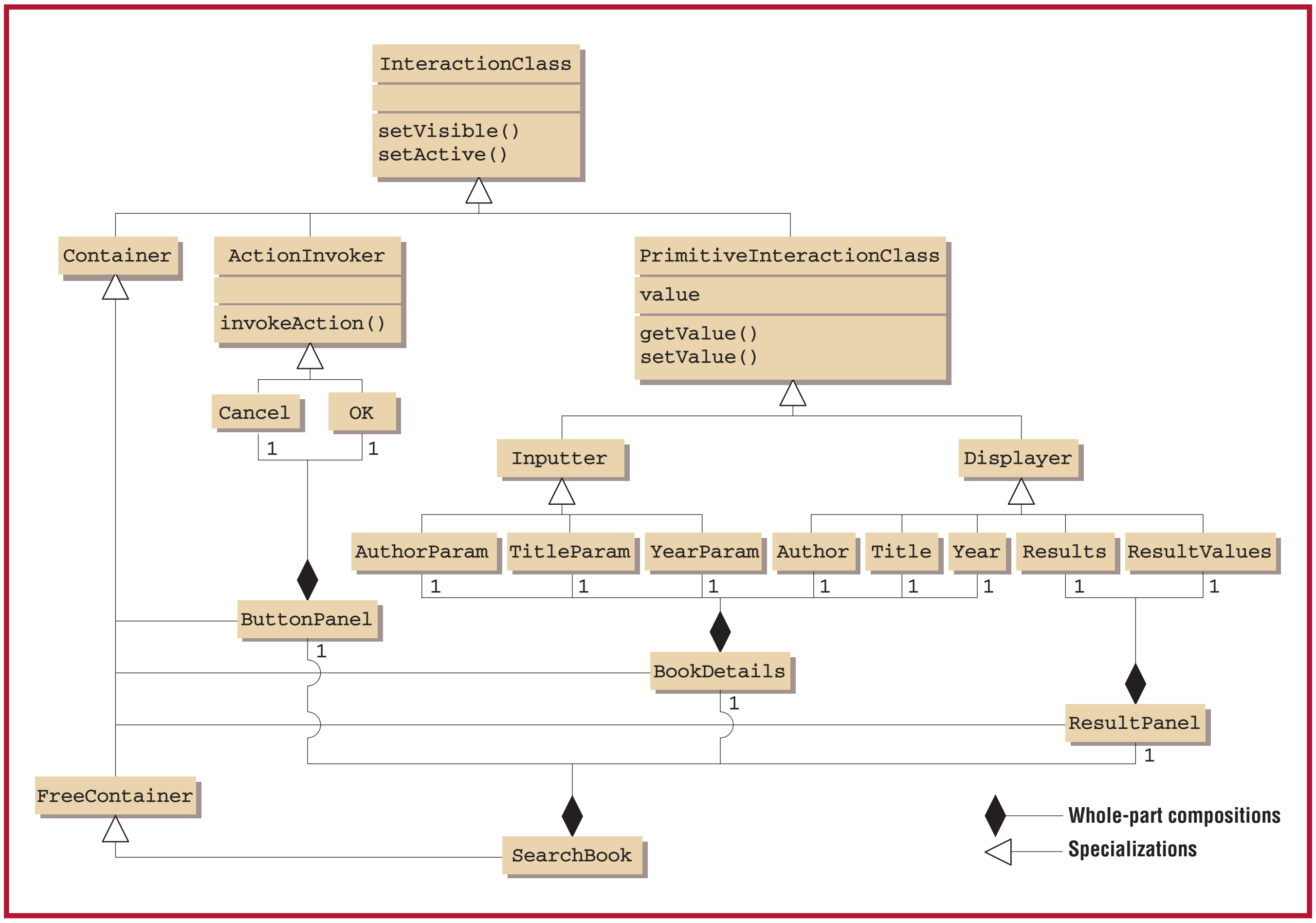


Figure 2. An abstract presentation model for the `SearchBook` interface. Numbers on associations represent cardinality constraints.

developing abstract user interface descriptions and have developed various models and notations, some of which are now used widely.[12] Specialist interface developers have expectations concerning different modeling facilities' suitability that general-purpose modeling techniques might not satisfy. Interface developers use several categories of models during development. You can represent the information such models capture using standard UML facilities.

Modeling interaction objects

A user interacts with a system through interaction objects. Interaction objects are commonly classified as either abstract or concrete. A concrete interaction object, any widget, is a physical implementation of an abstract interaction object. For example, a menu and a combo box are both concrete examples of an abstract chooser interaction object because menus and combo boxes let users select an item from an offered collection. Most interface builders (such as Microsoft Visual C++) provide facilities for interactively selecting and placing concrete interaction objects during interface development. However, specifying specific concrete interaction object placement is very much an implementation activity. And specifying an interface using concrete interaction objects risks premature commitment to a specific look and feel.

Several research projects have tried to support both abstract and concrete interaction objects. For example, the Trident project uses production rules in selecting concrete interaction objects from information on features such as screen density, target user experience, and required precision of input values.[13] Because you typically characterize interaction objects by their stored properties and the operations they support, you can use class diagrams in their description. Figure 2 shows an abstract description of the library search form from Figure 1. In this abstract presentation model, `InteractionClass` represents different functions that typify those supported by concrete interaction objects in widely available widget sets, and `Container` represents the grouping of components and containers in an interface.

You can represent a corresponding concrete presentation model as UML classes. You might reverse-engineer a UML class diagram from an existing object-oriented widget set, such as Java Swing, as a practical option. You could then allocate concrete widgets to support abstract components with a UML framework, as described in Figure 3. In Figure 3, the class diagram in Figure 2 is the specification of the pattern represented by the `SearchBook` presentation framework collaboration. You can bind concrete classes to abstract ones using the `SearchBook` presentation framework.

You can represent important features of both concrete and abstract presentations using standard UML class diagrams. Additionally, you can describe interaction objects' associated behavior using standard UML sequence or activity diagrams. However, the interface description shown in Figure 2 doesn't give much of a feel for the functionality or organization of the associated interface shown in Figure 1. Indeed, representing interfaces using class diagrams can quickly become obscure, a problem that has led to proposals for specialized abstract presentation models for UML. UML*i* also provides specialized visualizations for abstract presentation models. However, UML*i* interface diagrams are essentially UML class diagrams that clarify the purpose of individual abstract components and the containment relationships between different components.

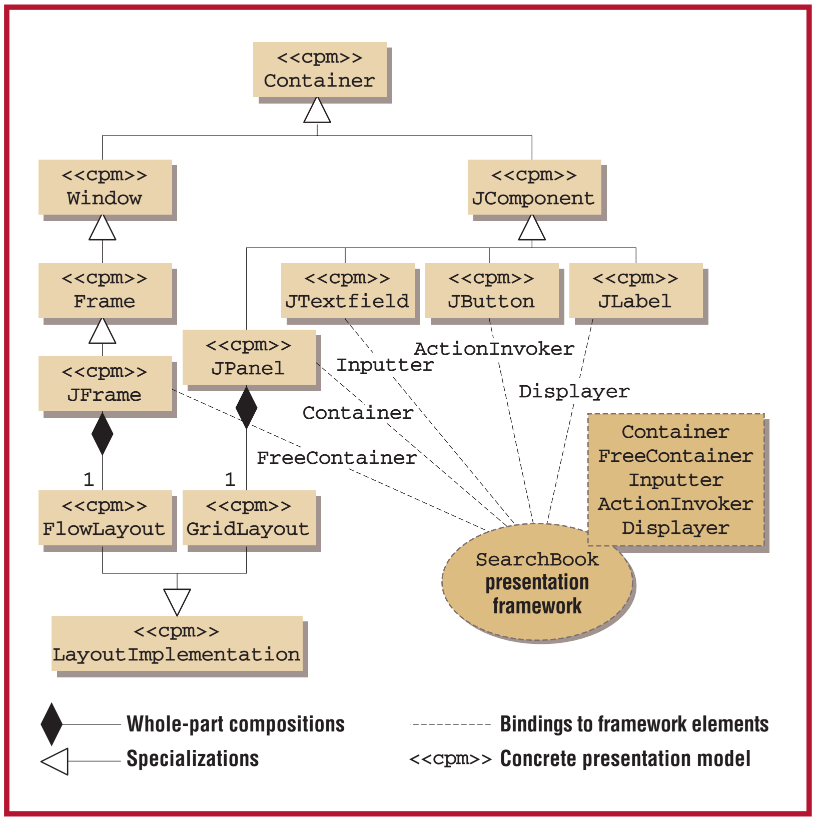


Figure 3. Using a UML framework to associate abstract and concrete interaction objects. Numbers on associations represent cardinality constraints.

Modeling tasks

Providing abstract task descriptions is central to most MB-UIDEs.[14,15] You generally represent a task model as a tree in which leaf nodes are primitive tasks and nonleaf nodes group and describe relationships between their children nodes. The following features are common to many task models:

- *Hierarchical decomposition.* High-level tasks systematically decompose into less abstract tasks.
- *Temporal relationships.* The order in which a composite task's children are carried out depends on the parent's temporal relation.
- *Primitive tasks.* The lowest-level nodes described in the task model are primitive tasks. An action task, for example, corresponds to an activity the application carries out. An interaction task involves some degree of human-computer interaction.

In practice, two schools of thought exist concerning task models and their use. One school holds that a task is a design artifact that elicits goals and their subgoals. The other holds that a task is a design artifact that describes the execution of actions representing subgoals. By performing such subgoals, users can achieve their goals when interacting with the application. In this second approach, tasks can describe actions, potentially to a precise level.

Whatever position you take, most researchers accept the effectiveness of high-level

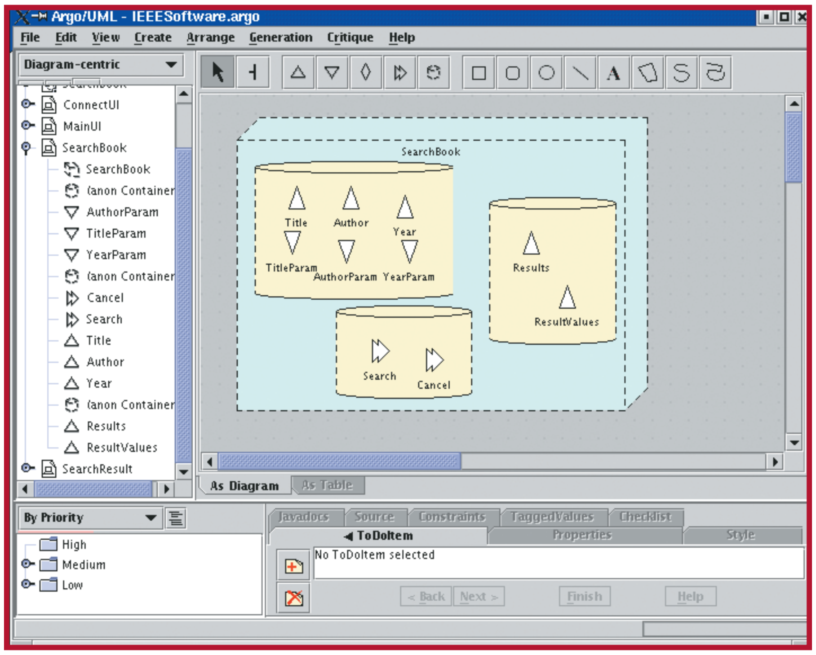


Figure 4. The UML*i* user interface diagram for book searching.

task-modeling facilities as part of user interface design. Moreover, most researchers investigating the extension of UML to support user interface modeling propose some form of task model. In UML*i*, rather than directly incorporating a new task model, you can use activity diagrams to model temporal relations along the lines of those supported by task models.

Relationships between models

Two principal forms of relationship exist between models in MB-UIDEs: the role of models in interface design and the explicit relationships between model components. The first form indicates the order that models are constructed and the way the population of one model informs that of another. A design method often supports this form of relationship. The second form represents the relationship between model components explicitly in the models, which indicate the places where responsibility for describing different aspects of an application's functionality passes from one model to another.

Both forms of relationship contribute to a design activity's success, but we focus on the second. MB-UIDEs typically describe the application you're constructing an interface for using a domain model. You usually use a conceptual database model, such as an entity-relationship model or an object model, for the domain model. You specify the links between domain and task model concepts at an abstract level in many MB-UIDEs. However, in the UML setting, you can model the application and the interface in a uniform setting, so you should describe the relationships between interface and application models precisely. We see little evidence that other proposals for user interface modeling in UML have addressed this issue in detail for their interface extensions to UML.

Modeling user interfaces in UML*i*

Because you can model abstract and concrete interaction objects using class diagrams, no particular need seems to exist to extend UML's representational facilities to describe interface components. However, class diagrams don't necessarily provide an intuitive interface representation. UML*i* provides an alternative diagram notation for describing abstract interaction objects. Figure 4 shows the user interface diagram for the abstract presentation model of the `SearchBook` interface in Figure 2.

UML*i*'s user interface diagram consists of six constructors:

- `FreeContainers` rendered as dashed cubes. A `FreeContainer` is a top-level interaction class that no other interaction class can contain.
- `Containers` rendered as dashed cylinders. A `Container` is a mechanism that groups interaction classes other than `FreeContainers`.
- `Inputters` rendered as downward triangles. An `Inputter` receives information from users.
- `Editors` rendered as rhombi (not shown in the diagram in Figure 4). An `Editor` facilitates the two-way exchange of information.
- `Displayers` rendered as upward triangles. A `Displayer` sends information to users.
- `ActionInvokers` rendered as right-pointing arrows. An `ActionInvoker` receives direct instructions from users.

As Figure 4 shows, a user interface diagram differs from an actual interface display. However, the notation indicates interaction classes' grouping, containment, and purpose without committing to look-and-feel or layout details.

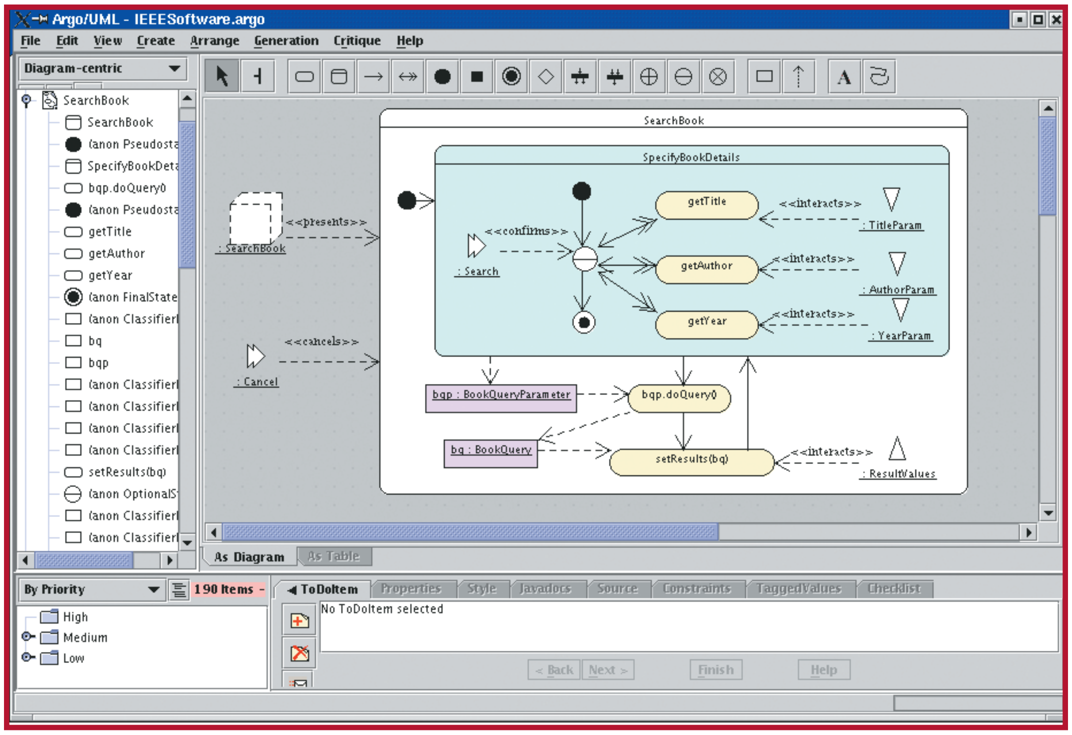


Figure 5. An activity diagram that models the `SearchBook` user interface's behavior.

The `InteractionClass` shown in Figure 2 is a subclass of the class constructor in the UML metamodel. In fact, the classes in Figure 2 that represent the user interface diagram constructors—`FreeContainer` and `Inputter`—are part of the UML*i* metamodel. You can map abstract interaction classes to concrete representations using different approaches, such as those used in Trident[13] or Teallach,[14] in which each abstract category associates with a collection of concrete representations from different styles.

Modeling tasks in UML*i*

Use cases and activities in UML*i* represent the notion of task, as it's conceptualized in the MB-UIDE community. Using use cases and their scenarios, you can elicit user interface functionalities required to let users achieve their goals. Using activities, you can identify possible ways to perform actions that support the functionalities elicited using use cases. Therefore, mapping use cases into top-level activities can help describe a set of interface functionalities similar to that described by task models in other MB-UIDEs.

Use-case diagrams in UML*i* are UML use-case diagrams. Activity diagrams in UML*i*, however, extend activity diagrams in UML. In fact, UML*i* provides a notation for a set of macros for activity diagrams that you can use to model behavior categories usually observed in user interfaces: optional, order-independent, and repeatable behaviors. The `SpecifyBookDetails` activity shown in Figure 5 uses the optional behavior, rendered as a circle overlaying a minus symbol. There, you can execute the activities `getTitle`, `getAuthor`, and `getYear`—which are called the optional behavior's selectable activities—any number of times, including none.

When needed, we render the order-independent behavior as a circle overlaying a plus symbol. Users interacting with the application can activate selectable activities or order-independent states on demand. Every selectable activity should execute once during the performance of an order-independent behavior. We render the repeatable behavior as a circle overlaying a multiplication symbol. Unlike the order-independent and optional behaviors, a repeatable behavior should have only one associated activity. You can specify a specific number of times that the associated activity should execute.

Using these macro notations, activity diagrams in UML*i* can cope better with the tendency that activity diagrams have to become complex even when modeling the behavior of simple user interfaces. Moreover, these notations might help interface modelers use

Object-oriented design metrics can quantify the inherent difficulties of constructing and understanding models because they measure many dimensions of the model's complexity.

activity diagrams. In the case study, we used the optional behavior to model the `SearchBook` interface's behavior, as Figure 5 shows. In the example, users can specify a combination of a book's title, author, and year before finishing the optional selection state using the `Search ActionInvoker`. Leaving the optional selection state, the application also leaves the `SearchBookDetails` activity, which eventually results in submitting the search query.

Relationships between models in UML*i*

We use object flows in activity diagrams to describe how to use class instances to perform actions in action states. In fact, by using object flows, you can incorporate the notion of state into activity diagrams that are primarily used for modeling behavior. In UML*i*, you can also use object flows to describe how to use interaction class instances. However, object flow states—rendered as dashed arrows connecting objects to action states—have specific semantics when associating interaction objects to activities and action states. UML*i* specifies categories of object flow states specific to interaction objects:

- The `<<interacts>>` object flows relate primitive interaction objects to action states, which are primitive activities. They indicate that associated action states are responsible for interactions in which users invoke object operations or visualize the results of object operations.
- The `<<presents>>` object flows relate `FreeContainers` to activities and specify that the associated `FreeContainers` should be visible while the activities are active.
- The `<<confirms>>` object flows relate `ActionInvokers` to selection states and specify that selection states have finished normally.
- The `<<cancels>>` object flows relate `ActionInvokers` to composite activities or selection states and specify that activities or selection states have not finished normally and that the application flow of control should be rerouted to a previous state.
- The `<<activates>>` object flows relate `ActionInvokers` to other activities, thereby triggering the associated activities that start when an event occurs.

The activity diagram shown in Figure 5 exemplifies the use of most of these object flows. For instance, the `SearchBook FreeContainer` becomes visible and the `Cancel ActionInvoker` becomes active when the `SearchBook` activity is active. Then you can specify a book's properties when the `SearchBookDetails` activity becomes active. Because the selection state in `SearchBookDetails` is optional, the `Search ActionInvoker` is also enabled for interaction when the `SearchBookDetails` activity is active.

We modeled a library system in both UML and UML*i* as a case study to assess the benefits of using UML*i* to model an interactive system. The `SearchBook` functionality described here is one of that library system's nine functionalities.[16] We produced two sets of files containing a textual representation of the UML and UML*i* models using ArgoUML*i*. The size of the textual representations of the models and diagrams are 3.44 and 4.01 times higher in UML than in UML*i*. Size, however, doesn't usually say much about the difficulty of constructing an interactive system model or of understanding such models. By contrast, object-oriented design metrics can quantify the inherent difficulties of constructing and understanding models because they measure many dimensions of the model's complexity.

Measuring the metrics in the library system's models is straightforward. The major concern regarding the use of these design metrics is producing models using two different notations for a common set of properties from the system's specification. We therefore used a systematic mapping strategy[16] to ensure that we adopted the same reuse strategy in both models.

In terms of structural complexity, our study demonstrated—by measuring and analyzing the suite of Chidamber-Kemerer (CK) metrics[17]—that we can achieve reductions in structural complexity in UML*i* models. In particular, we achieved a significant reduction of 87 percent in Response For a Class—defined as the number of methods executed in response to a message received by an object of that class—in UML*i* models. Indeed, introducing the interaction object flow with its stereotypes has simplified action modeling related to user interface widgets.

In terms of behavioral complexity, we measured and analyzed McCabe's cyclomatic complexity[18]—defined as the number of decisions (or predicates) specified in models plus one—and found we achieved a considerable reduction of 14 percent in cyclomatic complexity in UML*i* models. By introducing selection states, we simplified modeling of behavior commonly observed in user interfaces. These metric improvements indicate that constructing and maintaining interactive system models should be simpler and easier in UML*i* than in UML.

About the Authors

Paulo Pinheiro da Silva is a postdoctoral fellow at Stanford University. His research interests include conceptual modeling methodologies and tools and languages for modeling and verifying software systems. He is working on the development of tools for the Semantic Web. He received a PhD in computer science from the University of Manchester. Contact him at pp@ksl.stanford.edu; www.ksl.stanford.edu/people/pp.

Norman W. Paton is a professor of computer science and colead of the Information Management Group at the University of Manchester. His research interests include active, spatial, and deductive object-oriented databases and user interfaces to databases. He is working on spatiotemporal databases, distributed information systems, grid data management, and information management for bioinformatics. He has a PhD in computing science from Aberdeen University. Contact him at norm@cs.man.ac.uk; www.cs.man.ac.uk/~norm.

References

1. P. Pinheiro da Silva and N.W. Paton, "User Interface Modeling with UML," *Information Modeling and Knowledge Bases XII*, H. Jaakkola, H. Kangassalo, and E. Kawaguchi, eds., IOS Press, 2001, pp. 203–217.
2. P. Markopoulos and P. Marijnissen, "UML as a Representation for Interaction Designs," *Proc. Australian Conf. Computer-Human Interaction*, CHISIG, 2000, pp. 240–249.
3. N.J. Nunes and J. Falcão e Cunha, "Wisdom: A Software Engineering Method for Small Software Development Companies," *IEEE Software*, vol. 17, no. 5, Sept./Oct. 2000, pp. 113–119.
4. F. Paternò, "Towards a UML for Interactive Systems," *Proc. 8th IFIP Working Conf. Eng. for Human-Computer Interaction* (EHC 01), Springer-Verlag, 2001, pp. 7–18.
5. F. Bodart and J. Vanderdonckt, "Widget Standardisation through Abstract Interaction Objects," *Advances in Applied Ergonomics*, USA Publishing, 1996, pp. 300–305.
6. S. Wilson and P. Johnson, "Bridging the Generation Gap: From Work Tasks to User Interface Designs," *Computer-Aided Design of User Interfaces*, F. Bodart and J. Vanderdonckt, eds., Namur Univ. Press, 1996, pp. 77–94.
7. B.A. Myers et al., "The Amulet Environment: New Models for Effective User Interface Software Development," *IEEE Trans. Software Eng.*, vol. 23, no. 6, June 1997, pp. 346–365.
8. P.A. Szekely, "Retrospective and Challenges for Model-Based Interface Development," *Computer-Aided Design of User Interfaces*, F. Bodart and J. Vanderdonckt, eds., Namur Univ. Press, 1996, pp. xxi–xliv.
9. C. Wiecha et al., "ITS: A Tool for Rapidly Developing Interactive Applications," *ACM Trans. Information Systems*, vol. 8, no. 3, July 1990, pp. 204–236.
10. J. Conallen, *Building Web Applications with UML*, Addison-Wesley, 2002.
11. J.E. Robbins, D.M. Hilbert, and D.F. Redmiles, "ARGO: A Design Environment for Evolving Software Architectures," *Proc. Int'l Conf. Software Eng.* (ICSE 97), ACM Press, 1997, pp. 600–601.
12. B.A. Myers, S.E. Hudson, and R.F. Pausch, "Past, Present, and Future of User Interface Software Tools," *ACM Trans. Computer-Human Interaction*, vol. 7, no. 1, 2000, pp. 3–28.
13. J.Vanderdonckt and F. Bodart, "Encapsulating Knowledge for Intelligent Automatic Interaction Object Selection," *Proc. Conf. Human Factors in Computing Systems* (INTERCHI 93), ACM Press, 1993, pp. 424–429.
14. T. Griffiths et al., "A Model-Based User Interface Development Environment for Object Databases," *Interacting with Computers*, vol. 14, no. 1, Dec. 2001, pp. 31–68.
15. A.R. Puerta, "A Model-Based Interface Development Environment," *IEEE Software*, vol. 14, no. 4, July/Aug. 1997, pp. 40–47.
16. P. Pinheiro da Silva and N.W. Paton, *Improving UML Support for User Interface Design: A Metric Assessment of UMLi*, tech. report KSL-02-04, Knowledge Systems Lab., Stanford Univ., Stanford, Calif., 2002.
17. S.R. Chidamber and C.F. Kemerer, "A Metric Suite for Object-Oriented Design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, 1994, pp. 476–493.
18. T.J. McCabe and C.W. Butler, "Design Complexity Measurement and Testing," *Comm. ACM*, vol. 32, no. 12, Dec. 1989, pp. 1415–1425.