Components of a Collaborative Application

This is more decomposition than what may be necessary. This document was written in a real hurry and both the writing and the contents probably have lots of problems.

1. Model

Defines a set of fields and relationships among these fields. The semantics of these fields are not dependent on any of the objects described below. Example: program (e.g attributed syntax tree), spreadsheet, document, outline, text buffer.

The generic model (top-level model class) supports methods to get fields, set fields, get methods, invoke methods (Java reflection model, Sid, Munson interface). It should probably also indicate dependencies among fields, information that can be used by concurrency and access control components.

2. HyperModel

A model that defines links. Indicates which fields are structure fields and which are hyper links (ala DGS).

3. Database/Views

A *view* is an object that selects, transforms, permutes the data from another object, called its *database*, and also is editable in that it can be used to change the underlying database. It is like a database view except that it is editable and defined for objects rather than relational databases. Database views can themselves be databases for other views, so we can have cascaded views. Example of database, view pairs: budget/sorted budget, complete budget/direct costs,

A database provides the same interface as a model. In addition, it is associated with ‘‘aspects’’, which can be considered sets of fields. It is also an observable, and can send messages to observers.

A view is an observer. In addition, it should probably indicate which aspects of its database it is dependent on.

4. View Binder

Decides what view objects are bound to a database object based on user-defined view specification. Example: determines if sorted-budget or direct-cost views are connected to budget.

5. Abstractions/Interactors

An *interactor* is an object responsible for providing a user interface to modify another object called its *abstraction*. It is responsible for rendering/visualizing the state of the abstraction and also providing a set of commands to manipulate the latter. It is like a toolkit widget, though it may be defined by any user interface layer (window system, toolkit, user interface management system.)

An abstraction may be a model or another interactor, thereby allowing cascading of interactors. Examples of abstraction/interactor pairs: integer/slider, program/parse tree editor, parse tree editor/text editor, text editor/window, slider/window, budget/table editor, table editor/window.

An interactor not only renders or transforms the information in an interactor, it also adds some syntactic sugar. E.g: parse tree adds ‘‘if’’ and ‘‘then’’, sliders and dialog boxes add scrollbars, and windows adds pointers. While transformed information depends on the underlying abstraction, the syntactic sugar does not and can be changed without interacting with the abstraction.

An abstraction provides the same interface as a model. An interactor indicates which of its fields are dependent fields (that is renderings/transformations of abstraction fields) and which are syntactic sugar. It is an observable.

Abstraction/interactors are like databases/views except that the information in an abstraction is truly an abstraction of (that is, ‘‘higher level’’ than) the information in an interactor and also an interactor adds syntactic sugar while a view does not. Information in the interactor (abstraction) must be parsed (unparsed) to derive the information in an abstraction (interactor). A smalltalk view can be considered a combination of a database view and an interactor.

6. Parser/Unparser

They connect abstractions and interactors. A parser translates changes broadcast by an interactor before sending them to the abstraction and an unparser translates changes received from an abstraction before sending them to its interactor. E.g: a budget/text editor abstraction/interactor pair. Changes to fields of the budget are unparsed to changes to the text editor and similarly changes to the text editor are parsed into changes to the budget fields.

Parses/Unparsers may be combined with in/out couplers. They may be associated with interactor binders. Example of generic parser/unparser pairs: Consider a a Suite-like structure editor that traverses the fields of an arbitrary object and creates a tree of text/slider interactors based on the types of the leaf level nodes. A generic parser/unparser can be written for such an editor.

7. Interactor Binder

Binds abstraction and interactor objects with associated parsers/unparsers. Example Suite dialogue manager binding text/slider objects to a hierarchical object.

8. Peers

So far we have seen asymmetric relations among objects: database/view and abstraction/interactor. We an also have symmetric relations wherein versions/replicas of objects are created with some rules for synchronization. Versions/replicas can be considered *peer* objects. Peer are both observers and observables, are of the same type (look the same externally) and may even be of the same class (have the same internal representation.) Example: two replicas of the budget in two processes, each process providing local reads of the budget to its users. Peers essentially implement distributed shared memory.

9. Representations

Like peers except that they provide a different interface/implementation of the same abstract object. Example, fractional vs absolute representation of the budget. They need translators between them to exchange information among them.

10. In/Out Transformers

They connect representations of the same object and are responsible for converting to/from a common representation. Similar to couplers and in/out parsers. May be combined with couplers.

11. In/Out Couplers

They couple peers/databases/representations/abstractions to peers/views/representations/interactors (possibly in conjunction with parsers, unparsers, in/out transformers). They allow us to to separate the connection logic from the objects being coupled. Since the connection may be asymmetric (even between symmetric objects) we have separate in and out couplers. A connected object announces all communicable events to its out coupler and receives announced events from its in coupler. The in and out couplers determine what is actually transmitted/received to/ from other objects. For a pair of connected objects, two in/out coupler pairs are created, one pair for each object.

One can use the Field Message server as a basis for implementing generic in/out couplers. The out coupler simply sends all events to other connected in couplers. The behavior of an in coupler is determined by a user-defined Field pattern that describes which events are to be received.

Other generic forms of in/out couplers: An out coupler that batches events it receives from its coupled object and sends them with a certain user-defined period instead of sending them incrementally. Similarly, an in coupler that batches events it receives from out couplers and delivers them periodically with a certain period to its coupled object.

We can also a more elaborate Suite-like in and out couplers. Assume that the coupled object indicates, with each event the field that changes and three change attributes, the structural level of the change (e.g. word, sentence, para, section), the correctness level of the change (e.g validated, tested, committed), and the time of the change. Each of these attributes is of an ordinal type, that is, takes values from an ordered set (say integer). The couplers is associated with user-defined parameters indicating the minimum values of these three attributes for events that are transmitted/received and also the fields that are to be transmitted/received. An event is not transmitted/received if the values of its three change attributes are not >= the minimum levels and its field is not one marked for sending/receiving.

An example of an application-specific in/out coupler is a budget coupler that only sends/receives changes marked as direct costs.

12. In/Out Merger

They merge concurrent changes made by different threads to a pair of coupled objects. An in merger receives changes from another object and merges them with the local object. An out merger receives changes from the coupled object and transforms them to a form understood by the remote in merger.

Example of in/out merger: Munson's merger based on change histories.

In/Out mergers can be combined with In/Out couplers.

13. Concurrency Controller

Associated with a controlled object. Determines if an operation invoked on a controlled object should be allowed based on the transaction of the operation.

14. Access Controller

Associated with a controlled object. Determines if an operation invoked on a controlled object should be allowed based on the subject (user) invoking the operation.

15. Transactions

Capture groups of operations that must be invoked atomically. E.g: a transaction that groups together changes to salary and equipment. Associated with an ID/time stamp.

16. Subject

An entity whose authority need to be checked. Can be a user or role as in Suite access control. Example: PI/researcher. The former can change the budget but the latter can view it.

17. User

Identifies a users of the system. Keeps finger like info with users.

18. Role

An association between a role name and a set of users or other other roles which *take* the role being defined. Example, a role Student, which may be taken by the user Joe or by the role PhDStudent. Can be considered a user group. A user can take several roles simultaneously.

19. Proxy

A proxy of a real object is another object that looks from the outside like the real object and is a representative to the outside world. Any operation invoked on/by the real object goes through its proxy. It can perform several tasks: Check with concurrency/access controllers when operations are invoked on/by the real object. It can be the one that decides when NotifyObservers is called. It can also mark changes for out couplers with the various levels and remove marked changes received from in couplers. It can provide a central registry that keeps track of the objects associated with the real object. For instance, it can provide a method that returns the concurrency controller of the real object.

Examples of proxies: Distview proxies, XTV pseudo servers, COLA adapters. Generic proxies need to be either generated by the compiler or require the use of the reflection interface.

20. History Object

Logs changes to an object: used by undo/redo, replay, migration, and other components.

21. Undo/Redo Object

Processes undo/redo commands and consults history object.

22. Sessions

Groups together a set of users, workstations, and shared objects. Each user who joins a session from a workstation gets a user-interface on his workstation to interact with the shared objects. Users and objects may be dynamically added removed from a session.

23. Session Directory

Keeps track of the sessions created, and like a Unix directory, may be hierarchical.

24. Many to One Relation

In many of the cases above, the relations are many-to-one (and maybe many-to-many) rather than one-to-one. For instance, a watch dial interactor might be bound to multiple abstractions: hour, minute, and second objects. A view object may do a relational join among multiple database table objects. A controller object may control several controlled objects. A peer may have a group of peers that are considered a single logical unit (e.g. budgets of the CB group).

25. Hierarchical Relations

Each object roots a tree of nodes, where a node is a direct or indirect descendent of that object. We may want to treat the object as a flat structure (consisting only of its direct fields) or as a hierarchy By treating it as a hierarchical structure, we can communicate fine-grain changes, create a structure editor interface, and perform fine-grained access and concurrency control.