

**COLLEGE OF COMPUTING
GEORGIA INSTITUTE OF TECHNOLOGY
TECHNICAL REPORT: GIT-CC-02-37**

DYNAMO Design Guidebook

June 27, 2002

The DYNAMO project is concerned with the assembly of components of interactive systems. It includes a design method, described in this guidebook, and a set of tools that support it. The DYNAMO design method starts with a declarative model of the assembly expressed using a graphical UML CASE tool. From the declarative model, DYNAMO tools automatically generate C++ wrapper classes that “glue” the components together. The DYNAMO design method comprises three phases that refine a conceptual model of a proposed assembly into interrelated components organized into layered mode components. In Phase 0, the environment in which the assembly executes is described in terms of external actors, the assembly itself, the communication among them, and the behavioral properties that the assembly guarantees to maintain. Phase 1 asks the designer to partition the assembly into its constituent components and their relationships, assigning responsibility for external actions and guarantee-maintenance to the components appropriately. Finally, Phase 2 asks the designer to layer the constituents as mode components, where lower-level components communicate status changes upward, and higher-level components make specific service requests of lower-level components. For each phase, the guidebook provide a purpose, a diagrammatic representation that describes the resulting design artifact, a set of steps to create that diagram, and a set of guidelines or design rules for making appropriate design decisions. Each phase is illustrated using the example of a simple text browser assembly.

Acknowledgements

This guidebook was written by the members of the DYNAMO project at Georgia Tech and Michigan State University. Authors were Corinne McNeely, Spencer Rugaber, Kurt Stirewalt, and David Zook.

Effort sponsored by the Defense Advanced Research Projects Agency, and the United States Air Force Research Laboratory, under agreement number F30602-00-2-0618.

DYNAMO Design Guidebook

Software *components* are units of software that must be assembled to create a software system. Components can come from different sources: They may already exist, in either source or binary form, or they may need to be custom-constructed for the system. How to assemble these component systems both correctly and efficiently is an important problem facing software architects today.

The DYNAMO project is concerned with the assembly of components of interactive systems. It includes a design method, described in this guidebook, and a set of tools that support it. The DYNAMO design method starts with a declarative model of the assembly expressed using a graphical UML CASE tool. The abstractness of declarative models permits concise representations and enables formal reasoning. From the declarative model, DYNAMO tools automatically generate C++ wrapper classes that “glue” the components together. To support efficiency and reuse, components are assembled using a layered, implicit-invocation architecture called a *mode-component architecture*. A *mode component* is a specialized component that alerts its clients when its state changes. Additionally, the correctness of these generated assemblies can be verified either statically, using tools such as theorem provers or model checkers, or dynamically, by run-time assertion checking.

The DYNAMO design method comprises three phases that refine a conceptual model of a proposed assembly into interrelated components organized into layered mode components. In Phase 0, the environment in which the assembly executes is described in terms of external actors, the assembly itself, the communication among them, and the behavioral properties that the assembly guarantees to maintain. Phase 1 asks the designer to partition the assembly into its constituent components and their relationships, assigning responsibility for external actions and guarantee-maintenance to the components appropriately. Finally, Phase 2 asks the designer to layer the constituents as mode components, where lower-level components communicate status changes upward, and higher-level components make specific service requests of lower-level components.

The following sections overview the design method. For each phase, the guidebook provide a purpose, a diagrammatic representation that describes the resulting design artifact, a set of steps to create that diagram, and a set of guidelines or design rules for making appropriate design decisions. Each phase is illustrated using the example of a simple text browser assembly. At the end of the document, a glossary of all DYNAMO-related terms is provided.

Phase 0

Purpose: To specify the interfaces between the assembly and those aspects of the environment with which it interacts.

Representation: The representation produced by Phase 0 is similar to a Data Flow context diagram, but expressed using UML class diagram symbols. The Phase 0 diagram contains one rectangular class icon denoting the assembly and additional rectangles for each user, system, or data repository (collectively, *actors*) with which the assembly interacts. Actors send requests (*events*) to the assembly, and the assembly responds with vis-

ual feedback (*percepts*). Actors sending events are connected to the assembly using directed lines; percepts of actors are assumed to be visible to other actors without explicit denotation. Assembly behavior is described by *guarantees* expressed as natural language annotations. The annotations can be associated either with event requests or with the assembly itself by using an undirected dotted line.

Process: Construct a UML diagram describing the assembly, the external actors with which it interacts, the percepts it provides, the events it responds to, and the properties it guarantees.

- Use a UML class symbol to denote the assembly as a whole. Give the assembly an appropriate name.
- Use other class symbols to denote the external actors (users, sensors, data repositories and other systems) with which the assembly interacts.
- Supply directed lines from external actors into the assembly for each event the assembly responds to. The events should be named and may be parameterized.
- Denote assembly percepts as attributes of the assembly class icon.
- Use natural language to express any guarantees of assembly behavior. There are two types of guarantees: *responses* to events and *invariants* (relationships among assembly percepts). Guarantees are added to the diagram as text annotations in note boxes attached via an undirected dotted line to either the event directed lines (for responses) or to the assembly class rectangle (for invariants).

Guidelines:

- There is often a choice of whether to express a guarantee as an invariant or a response. In general, responses should be simple and immediate, for example, indicating what happens to the specific GUI element that a user interacted with. More elaborate effects on the assembly can usually be more naturally expressed as invariants.
- Break guarantees down into simple sentences, one annotation per sentence.
- Guarantees may include but are not limited to naming commitments, constraints on use, responses to events, invariants, and synchronization properties.
- DYNAMO does not yet support quality-of-service (*e.g.* performance) guarantees.
- Determine all visible aspects of the GUI that convey meaning to the user and model them as percepts.
- Percepts that are not explicitly mentioned in the response to an event are assumed to be unaffected by occurrences of that event unless changes in their values can be inferred from assembly invariants.
- Percepts are universally visible to all actors including the assembly.
- Events can be thought of as asynchronous and atomic with respect to assembly behavior.
- No internal state appears in Phase 0 models.

Example: In the following example, the user interacts with a simple text browser assembly consisting of a viewport in which a document is displayed and a scrollbar for control-

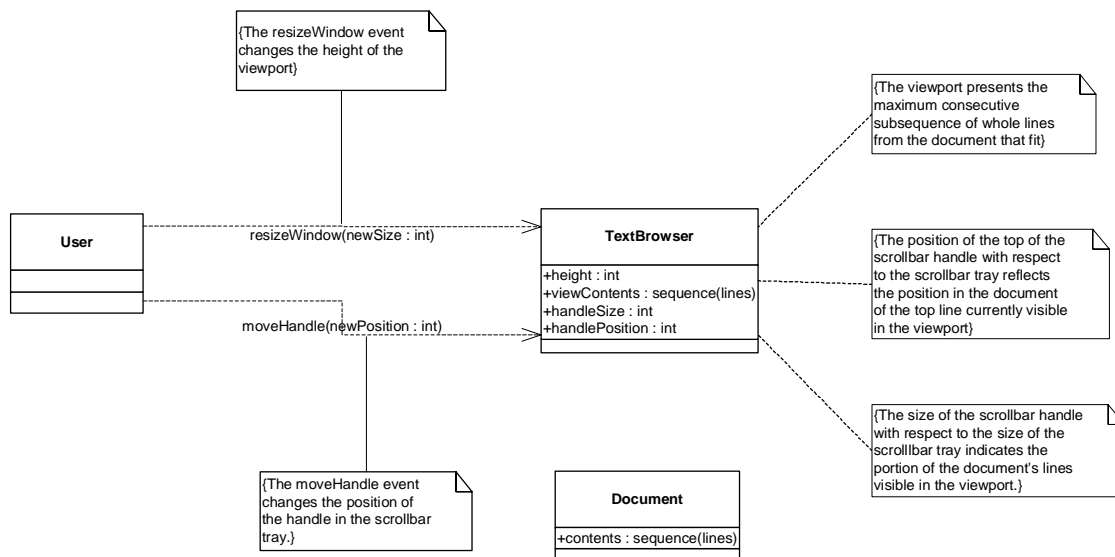
ling the lines viewable at any given time. The assembly contains four percepts (the viewport height, the lines displayed in the viewport, the position of the top of the scrollbar handle, and the size of the scrollbar handle relative to the size of the scrollbar tray). The assembly responds to two kinds of user events—resizing the viewport and moving the scrollbar handle—with the obvious response guarantees. In addition the assembly maintains the following invariants.

1. The viewport displays the maximal consecutive subsequence of complete lines from the document that fit within it.
2. The position of the top of the scrollbar handle relative to the scrollbar tray reflects the position in the document of the line currently visible at the top of the viewport. That is, moving the scrollbar handle allows different portions of the document to be displayed.
3. The size of the scrollbar handle with respect to the size of the scrollbar tray is equal to the number of lines visible in the viewport compared to the total size of the document.

Note that to simplify the exposition, we make the following assumptions about the assembly.

- All characters displayed in the viewport have the same point size;
- The viewport width is fixed at a value wide enough to display all characters in the widest line of the document.
- Other GUI events, such as moving the viewport and resizing the scrollbar have no effects on the percepts.
- The various ways in which the scrollbar handle can be repositioned (dragging, clicking, etc.) are combined into a single event.

Diagram:



Explanation: The diagram contains three actors: the TextBrowser in the center, the User on the left, and the Document in the bottom center. The TextBrowser is the as-

sembly being designed; the **User** and the **Document** are external actors. The **User** is active, issuing events to the **TextBrowser**. The **Document** is a passive data repository.

The **User** can initiate two events: `resizeWindow` and `moveHandle`. Each is annotated with a guarantee indicating the **User's** expectations for what happens to the assembly when the event is issued. The **Document** is an external actor with one percept, its contents. It issues no events. The presumption is that the contents are available for use by the assembly. The **TextBrowser** assembly itself is at the center of the diagram and contains four percepts: `height`, `viewContents`, `handleSize`, and `handlePosition`. The assembly invariant guarantees are expressed in note boxes on the right-hand edge of the diagram.

Phase 1

Purpose: To refine the Phase 0 assembly into components and to express assembly guarantees formally using UML's Object Constraint Language (*OCL*). The components must fit together in such a way that they provide the same behavior and presentation as the assembly described in the Phase 0 diagram.

Representation: The Phase 1 diagram is built using a subset of UML static model features. The components are represented as UML classes. The interactions among components are denoted with associations. Guarantees are expressed in OCL, either as pre and post conditions on event-handler operations (for responses), as invariants on the states of components, or as annotated associations among the components.

Process: Construct a Phase 1 diagram in the following manner.

- Break the Phase 0 assembly into components. Components can denote existing software from legacy systems, GUI toolkits, etc., or they may indicate components that are not yet written.
- Assign Phase 0 events and percepts to individual components:
 - Provide an operation in a component if that component is responsible for handling a given external event.
 - Provide an attribute in a component if that component is responsible for providing a percept.
- Express Phase 0 guarantees more formally using OCL constraints.
 - Response guarantees should be expressed as **pre** and **post** conditions on event-handler operations.
 - Maintaining an intracomponent invariant becomes the responsibility of that single component. For such an invariant, annotate the component with an OCL (**inv**) constraint that references only the component's operations or attributes.
 - Some invariants describe complex relationships among two or more components. In a case such as this, provide an association among the related components and annotate it with an OCL (**inv**) constraint expressing the invariant.
- If necessary iterate the Phase 1 process, refining until all guarantees are adequately expressed.

- During refinement, a percept may be decomposed such that several components contribute to providing the information it expresses.
- Likewise, an event handler can delegate parts of its responsibility to subordinate operations.

Guidelines:

- Only those component features (operations and attributes) that are relevant to the assembly's external behavior (its interface and guarantees) are included in the Phase 1 diagram, even if more is known about an existing component.
- However, even though event handlers are normally simple and self-contained, sometimes part of an event response can be naturally delegated to operations in other existing components. In these cases, the delegatee operation may be explicitly expressed in the appropriate component, the delegator **pre** and **post** conditions can refer to the delegatee, and the delegatee can be annotated with appropriate OCL (**pre** and **post**) constraints.
- In the course of expressing the OCL constraints, they should be given mnemonic names.
- Sometimes a percept of an external actor is of interest to an assembly. It may be useful in situations like this to introduce a component into the assembly that acts as an interface to that actor, thereby enabling the DYNAMO tools to directly generate code for ensuring assembly guarantees related to those percepts. Once these sorts of components have been introduced, there is no longer a need to include external actors in the Phase 1 diagram.

Example: In the Phase 1 diagram of the text browser, the assembly is broken down into ViewPort, ScrollBar, and FileManager components. First, Phase 0 events and percepts are allocated to components. The ViewPort component is responsible for displaying the Document's contents. Assigned to it are the height, and viewContents percepts and the resizeWindow event handler. The ScrollBar component interacts with the User actor to control which lines are actually displayed. The ScrollBar is responsible for the handlePosition and handleSize percepts and for handling the moveScrollBar event. The FileManager component is responsible for interacting with the Document external actor.

The Phase 0 guarantees are expressed with the following OCL constraints.

context ScrollBar::moveHandle(newPosition : int): void
post : handlePosition = newPosition

context ViewPort::resizeWindow(newSize : int) :void
pre : newSize >= 0
post : height = newSize

context displaysDocument **inv**:
 ViewPort::viewContents =
 FileManager::document->subsequence(ScrollBar::handlePosition, ScrollBar::handlePosition + ViewPort::height - 1)

context scalesHandle **inv**:
 ScrollBar::handleSize = ViewPort::height / FileManager::document->size()

context linesVisible **inv:**

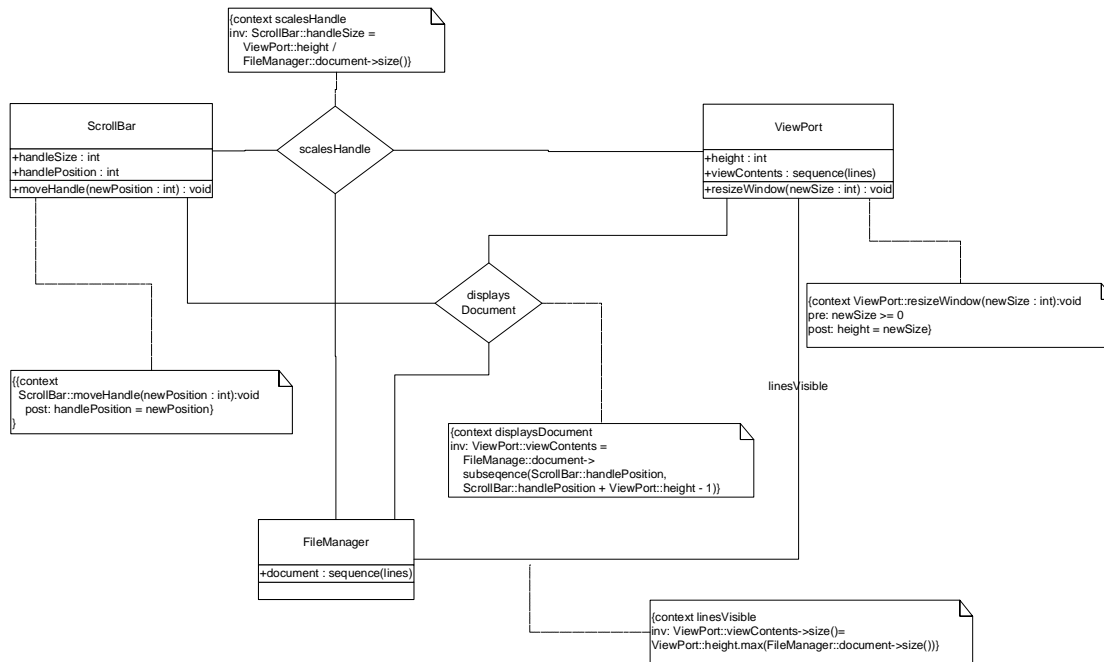
```
ViewPort::viewContents->size() =  
ViewPort::height.max(FileManager::document->size())
```

Several more simplifying assumptions have been made to clarify the exposition.

- Attributes of all three components are defined in units of (an integer number of) lines. For real GUI toolkit components, this might not always be the case. For example, the **ScrollBar** or the **ViewPort** might be expressed in terms of screen pixels. Translations among these units would be straightforward to express in OCL.
- Maintaining invariants, such as **linesVisible**, when the user has clicked in the tray above the handle to move the **ViewPort** contents up by the number of lines that the **ViewPort** can display, requires more elaborate constraints. These constraints must deal with issues such as what happens when the currently viewed top line is less than **height** lines from the top of the **Document**. None of these are difficult to express, but they do crowd the diagram.
- There are some special cases needed to deal with computing the size of the handle. For example, what should be the size of the handle when the document is empty? These special cases can be included in the OCL using **max** and **min** operators.
- Not every position in the scrollbar tray can be occupied by the top of the handle. That is, because the handle has a non-zero height, that portion of the bottom of the tray is not available for positioning. And, because the scrollbar's handle changes size when the viewport is resized, the size of the excluded region changes during the course of the assembly's execution.
- Because the specification requires that the top line in the **ViewPort** is fully displayed, the **ScrollBar** handle position computation must result in an integer value.
- In creating the Phase 1 constraints, a potential inconsistency in the Phase 0 guarantees becomes apparent. The first guarantee indicates that a maximal subsequence of lines is displayed in the viewport. The second guarantee implies that any line in the document can appear as the top line of the viewport. So what happens when the scrollbar handle is moved as far as possible downward in the scrollbar tray. The implication is that the last line of the document should appear as the top line of the viewport. But then it would be the only line visible, and this is not a "maximal subsequence" of the document. The problem can be resolved by interpreting "maximal subsequence" as "maximal possible subsequence from the top line toward the end of the file". Note that it is possible to build text browsers in which the user can only scroll downward until the last line of the document appears as the last line of the file or in which the user can continue to scroll past the last line in the document so that no lines are visible at all. Detection of inconsistencies such as this should be considered a feature of using formal methods rather than a problem.
- A related situation arises when a full viewport is enlarged so that there are insufficient lines remaining in the document to populate it. Some text browsers (such as Netscape) deal with this situation by repositioning the scrollbar handle in such a way that the resized viewport is fully populated. Other text browsers (such as

WordPad) treat the situation the same way we do, by leaving the top displayed line unchanged.

Diagram:



Explanation: The assembly has been broken into three components: ScrollBar in the upper left-hand corner, ViewPort in the upper right-hand corner, and FileManager in the bottom center.

The handleSize and handlePosition percepts have become attributes of ScrollBar, and the moveHandle event handler has been added as an operation to Scrollbar. Likewise, the height and viewContents percepts have been allocated as attributes of ViewPort, and the resizeWindow event handler has become one of its operations. FileManager contains one attribute, called document, to serve as a surrogate for the contents of the document being displayed.

For each of the assembly events, an OCL constraint has been added to the appropriate event-handler operation directly expressing the Phase 0 guarantee. Each of the Phase 1 invariants refers to multiple components. One possibility is to select a component to hold each invariant, using OCL navigation syntax to reference the attributes of the other components. Instead, the DYNAMO design method suggests adding associations to the diagram being constructed. Assignment of invariants to specific components takes place in Phase 2 of the DYNAMO method, and using associations in Phase 1 reduces the possibility of biasing the decisions that will be made at that time.

Note that two of the associations are ternary (involve three components). DYNAMO has borrowed the diamond symbol from Entity-Relation diagrams to denote these situations. Each of the associations is annotated with an OCL constraint to express the relevant guarantee.

Somewhat subtle is the fact that two of the Phase 0 guarantees have been combined in the constraint attached to the `displaysDocument` association. That is, Phase 0 contained one guarantee that expressed that the viewport displays parts of the document and another that related the scrollbar position to the top line displayed. These have been combined in Phase 1 to express the exact lines that are displayed. This combination can be thought of as a design refinement, reducing the number of invariants to be maintained and thereby making the design that much simpler.

There is one new invariant added to the diagram on the association `linesVisible` to deal with the Phase 0 specification inconsistency described above. The new invariant states exactly what to expect when the user has scrolled so that not all of the viewport is used for displaying lines from the document.

Phase 2

Purpose: To impose a layered, implicit-invocation (mode component) architecture on the assembly. The components are layered, and the interactions between components are restricted so that a component can only interact with other components in the same or adjacent layers. The mode component architecture makes the assembly simpler and more reusable by reducing the interactions among the components. The declarative dependencies expressed non-procedurally in Phase 1 as OCL constraints are reformulated in such a way that DYNAMO code-generation tools can provide constraint consistency in an efficient way without requiring any explicit (*i.e.* hand-coded) intervention in the components.

Representation: The diagram is a UML class diagram. Components are denoted by classes that reside in hierarchical layers. More than one component can constitute a layer. A single dashed directed line connects a layer with the layer immediately beneath it in the hierarchy. It should be interpreted to indicate that the superior component makes use of the inferior component.

Interactions upward in the hierarchy occur implicitly. That is, if a component is dependent on information in components in the next lower level, changes in the state of those inferior components are communicated upward without an explicit operation call being made. Elements of state whose changes are communicated implicitly upward are called *status variables*.

Interactions downward in the hierarchy, which can be used to request state changes, occur in a normal call–return fashion. That is, a component can make an explicit request to an operation in a component in the next lower layer. Such explicit requests are called *service calls*.

There are no associations in Phase 2 diagrams. Instead, the OCL constraint annotating each Phase 1 association is replaced by a mode component constraint. A *mode-component constraint* uses a restricted form of OCL expression syntax in which the constraint takes the form of an equation with a single variable on the left-hand side and a general OCL expression on the right-hand side. The left-hand side variable must be an attribute of a component in the superior layer; all other referenced variables must be attributes of variables in components of the inferior layer. The mode-component constraint is expressed as an annotation on the component to which the left-hand side variable belongs. The intended interpretation of a mode-component constraint is that a change to any

variable on the right-hand side causes the value of its expression to be recomputed and assigned to the variable on the left-hand side.

Process: Construct a Phase 2 diagram in the following manner.

- Components from the Phase 1 diagram are placed in layers. More than one component may be in a single layer.
- Component attributes whose values are referenced by components in the superior layer become status variables. A component's status variables are only visible to components in its own layer and in the layer immediately above it in the hierarchy.
- Remove associations. This may take place in several ways.
 - Ternary (and higher) associations are removed using normal OO design tactics, such as substituting several binary associations or introducing a new object (component) to implement the association.
 - Each of the remaining binary associations has an annotation indicating an assembly invariant. Reformulate the invariant as a mode component constraint by algebraically manipulating it.
 - If the variables on the right-hand side belong to components in a layer inferior to the component containing the variable on the left-hand side, then, remove the association, and place the mode component constraint as an annotation on the component containing the left-hand side variable of the mode component constraint.
 - If the left-hand side variable belongs to a component in a layer below the layer with the components containing the right-hand side variables, then create multiple new operations in the lower component, one for each variable on the right-hand side. The new operation's **post** condition should indicate the required update.
 - If the invariant is not an equation or if the left-hand side of the equation cannot be resolved to a single variable, it is still possible to designate superior and inferior components and their associated status variables. In this case, introduce an operation in the superior component named **update_X**, where X is the name of the superior status variable, and make its **pre** and **post** conditions reflect the OCL constraint. The DYNAMO compiler will guarantee that the operation will be called whenever the value of X changes.
 - The same remedy applies to situations where the OCL constraint is an equation, and the left-hand side is not a variable, but some derivative of one. An example of this kind of situation occurs in the example below, where the OCL constraint refers to the size of a collection rather than to the collection itself.
- An invariant that is the responsibility of a single component in the Phase 1 diagram is refined in a normal OO manner. That is, the invariant is established when the component is created, and each operation of the component is responsible for maintaining it.

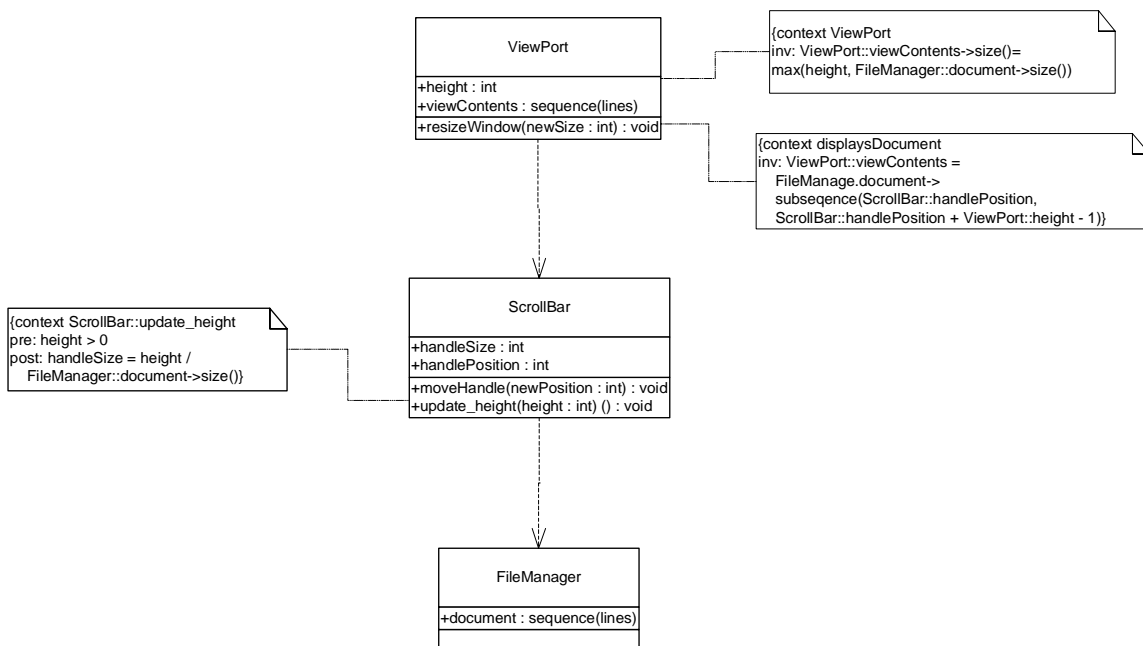
Guidelines:

- Component layering may be partially determined by the constraints on the associations between components. In general, layers should be organized in such a way that service requests flow downward and status updates flow upward. This normally means that user events are handled by components at the top of the layering. As they are handled, service requests are made downward. Then status updates flow upward until percepts at the top of the hierarchy present assembly status back to the user.
- If a mode component constraint references components in non-adjacent layers, it becomes necessary to introduce intermediate status variables and services into the hierarchy to propagate information in a layer-by-layer fashion. Fortunately, the DYNAMO OCL compiler can infer these intermediate features from the OCL navigation syntax, so the designer does not have to explicitly introduce them.
- In performing the algebraic manipulation required to transform a Phase 1 OCL constraint into a mode-component constraint, it may be necessary to break the constraint into several pieces. In general, a Phase 1 (binary) association invariant may refer to multiple interdependent attributes. It is the designer's job to reformulate it into a set of update dependencies. Situations where this is not possible often indicate design flaws. Even where it is possible, the designer still has to avoid update loops, in which two components cyclically update each other, potentially leading to infinite loops.
- Component interactions within a layer are handled in the normal, object-oriented design fashion.

Example: The Phase 2 diagram of the text browser assembly contains three layers: ViewPort on top, ScrollBar in the middle, and FileManager at the bottom. All of the Phase 1 attributes are promoted to status variables. Pre and post conditions are added to describe the services' behaviors.

Phase 1 event-response constraints are suitable for Phase 2 diagrams; so they have been omitted to simplify the diagram.

Diagram:



Discussion:

Both `VIEWPORT` and `SCROLLBAR` accept events and display percepts, so either could be at the top of the hierarchy. `VIEWPORT` was chosen because `VIEWCONTENTS` was considered to be a more important percept than `HANDLESIZE`. `FILEMANAGER`, handling no events, naturally is placed at the bottom of the hierarchy.

The three intercomponent association invariants were converted to mode component constraints as follows.

- The `DISPLAYSDOCUMENT` association invariant becomes a mode component constraint responsible for updating the `VIEWCONTENTS` attribute of `ViewPort`. Hence, it is associated with the `ViewPort` component.
- The `LINESVISIBLE` association invariant presents a problem: there is no variable to update. Instead the constraint expresses an invariant that concerns sizes. The invariant actually indicates what should happen to `VIEWCONTENTS` when height changes. As such, it naturally belongs in `VIEWPORT`.
- The `SCALESHANDLE` association invariant uses the `HEIGHT` attribute of `VIEWPORT` and the size of the `DOCUMENT` attribute of `FileManager` to reset the `SCROLLBAR HANDLESIZE` attribute. Hence, the correct placement is with `SCROLLBAR`. Because `HEIGHT` is an attribute in a superior component, an update operation must be introduced into `SCROLLBAR`. When `VIEWPORT::HEIGHT` is altered, a call must be made to the new operation to effect the update in `HANDLESIZE`. The post condition of the operation indicates the desired property.

Summary and Status

The `DYNAMO` project is exploring ways to build assemblies with guaranteed properties. This guidebook describes how a designer can refine high-level, natural language guarantees into operational constraints that the `COGITO` compiler can translate into efficient C++ wrapper code. A prototype of `COGITO` is nearing completion that will allow us to test both the generality of the approach and the efficiency of the generated code.

Several enhancements are currently being considered. For example, not all guarantees can naturally be expressed as equations. It is straightforward to compile a guarantee such as this into a simple operation that test the constraint and invokes a run-time exception if it is violated.

More interesting is the question of how to deal with guarantees that describe synchronization constraints that cross thread or process boundaries. We are designing a run-time infrastructure for dealing with status updates that cross these boundaries. The challenge is provide one with unacceptably compromising performance.

Glossary

- **actor:** A participant in the environment in which the assembly lives. Actors may be passive repositories of data or may proactively communicate with the assembly.
- **assembly:** A collection of software components that comprises a system.
- **component:** A unit of software assembled with other components to create a larger system.
- **event:** An atomic unit of communications, which may carry data.
- **guarantee:** A description of expected assembly behavior.
- **invariant:** A property of an assembly or a component that holds between event occurrences. Invariants are expressed in terms of relationships among the elements of the assembly's percepts or component's state attributes.
- **mode component:** A hierarchical software component whose interface provides a continuously updated view of its current status.
- **mode component architecture:** A layered, implicit-invocation architecture where all associated components are Mode Components
- **mode component constraint:** A single-assignment OCL expression where the left hand side contains a single status variable and the right hand side is a formula dependent upon status variables of mode components in an adjacent layer.
- **OCL:** The Object Constraint Language. Developed to express invariant, **pre** and **post** condition information in UML Diagrams.
- **OCL constraint:** An OCL expression attached to either a UML class or association.
- **percept:** Visual feedback; generally an attribute of the assembly that is visible to the user.
- **pre/post conditions:** Specific type of OCL expression that defines the behavior of a component service.
- **response:** A property of an assembly or a component that holds as the result of the assembly or component processing an event. Responses are expressed in terms of relationships among the elements of the assembly's or component's state.
- **service:** An explicitly invoked operation used in Phase 2 diagrams by a component in an adjacent state to alter a component's state.
- **status variable:** A Phase 2 component attribute that provides automatic notification when its state changes to client components in the adjacent layer.