Case Study
DYNAMO: Automatic Generation of Invariant Maintenance Wrappers

Spencer Rugaber

College of Computing

Georgia Institute of Technology

June 20, 2007

Further Information

- http://www.cc.gatech.edu/dynamo
- Kurt Stirewalt and Spencer Rugaber.
  "Automated Invariant Maintenance Via OCL
  Compilation."
  Lionel C. Briand and Clay Williams, editors,
  *Model Driven Engineering Languages and
  Systems,* Springer-Verlag, Lecture Notes in
  Computer Science, Number 3713, pp. 616-632,
  October 2-7, 2005, Montego Bay, Jamaica.

Architectural Design Process

- We have talked so far about various representations that can be used for expressing architectures

- But we haven't indicated yet what process might be used to actually perform the design

- The following case study presents one approach that combines top down and bottom up techniques

- Note that it makes use of UML to describe architecture in a way that could be enforced using metamodel OCL constraints

Case Study: Text Browser

- Consider the hypothetical situation where you must solve the following problem:
  - You have a source of textual data (Document) with a file system interface (FileManager)
  - You have a resizable viewing window resource capable of displaying lines of text (ViewPort)
  - You have a controlling device like a scrollbar capable of selecting a discrete value with the position of its handle and also of giving feedback via the size of the handle (ScrollBar)
- Objective is to specify the properties of the TextBrowser, choose an architecture and assemble the components
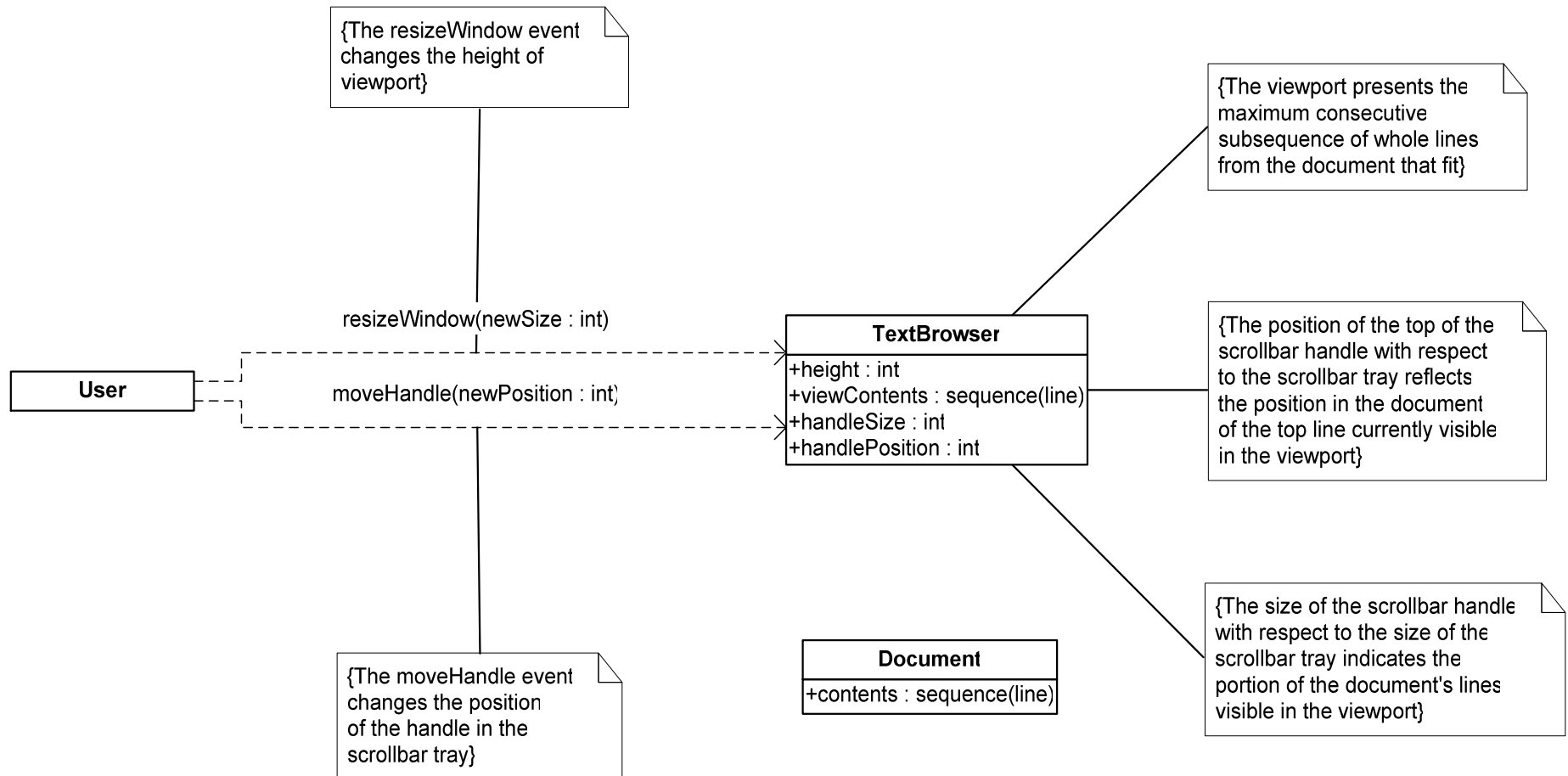
Phase0

- Construct a context diagram for the TextBrowser

- Indicate external actors but only one activity, the TextBrowser itself

- Indicate external stimuli (events) that can effect the TextBrowser

- Indicate how the TextBrowser communicates its results back to the external actors (percepts)

- Specify, in English, the properties you want the TextBrowser to have

TextBrowser Properties

- What properties would you expect a TextBrowser to have?
  - What events does it respond to?
  - How does the text that is displayed relate to the file contents?
  - What does the size of the ScrollBar handle tell you?

Phase 0

{The resizeWindow event changes the height of viewport}

{The viewport presents the maximum consecutive subsequence of whole lines from the document that fit}

resizeWindow(newSize : int)

TextBrowser

+height : int
+viewContents : sequence(line)
+handleSize : int
+handlePosition : int

moveHandle(newPosition : int)

User

{The position of the top of the scrollbar handle with respect to the scrollbar tray reflects the position in the document of the top line currently visible in the viewport}

{The moveHandle event changes the position of the handle in the scrollbar tray}

Document

+contents : sequence(line)

{The size of the scrollbar handle with respect to the size of the scrollbar tray indicates the portion of the document's lines visible in the viewport}
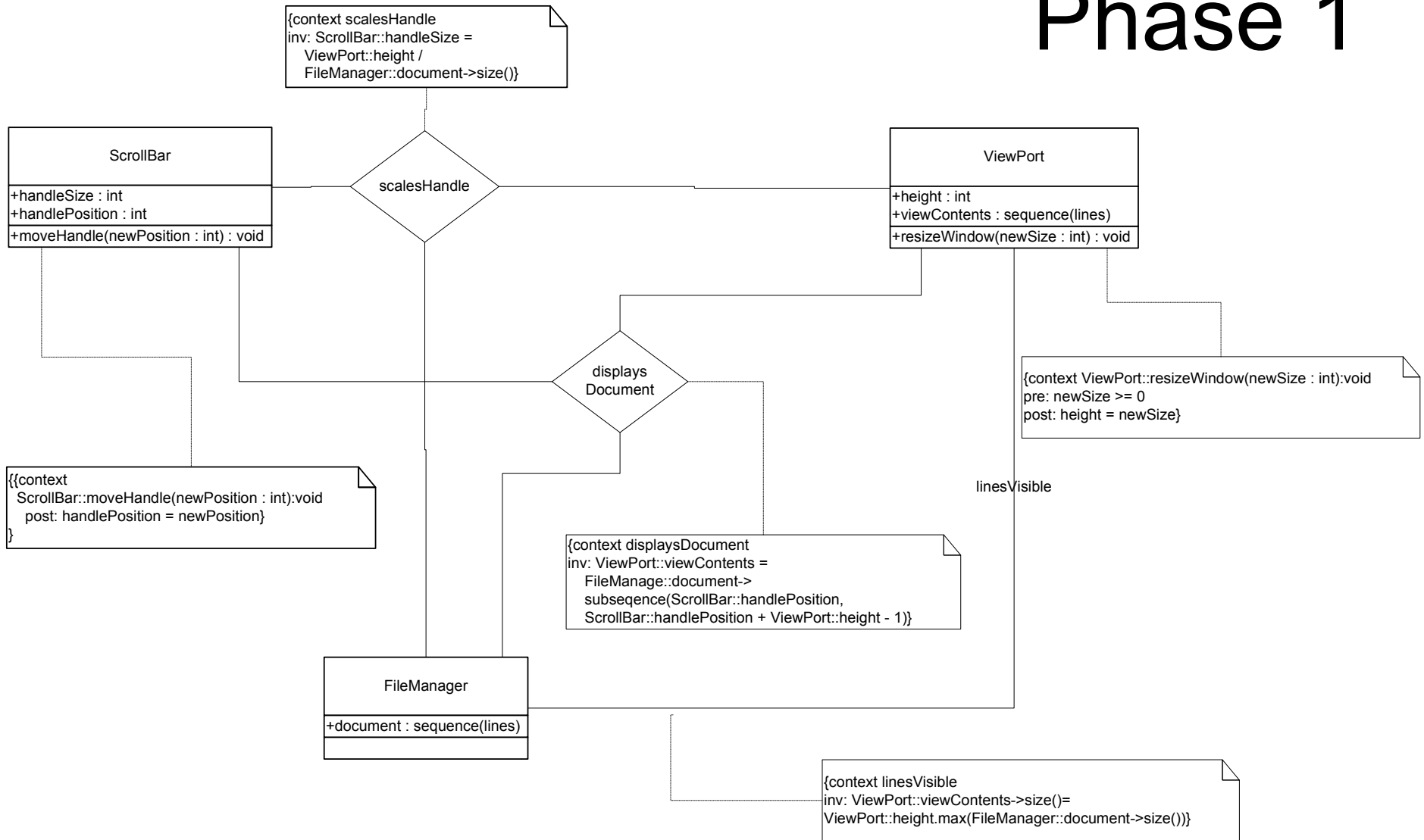
Phase 0 Guarantees

- The ViewPort displays the maximal consecutive subsequence of complete lines from the Document that fit within it.

- 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.

- 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.

© 2007, Spencer Rugaber

Phase 1

- Decompose system into components
- Allocate responsibilities (*invariant maintenance*)
  - Event handling
  - Percept delivery
  - Property guarantees
- Specification of component properties as OCL invariants and pre/post conditions

Phase 1

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

ScrollBar

+handleSize : int
+handlePosition : int

+moveHandle(newPosition : int) : void

scalesHandle

ViewPort

+height : int
+viewContents : sequence(lines)

+resizeWindow(newSize : int) : void

displays
Document

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

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

linesVisible

{context displaysDocument
inv: ViewPort::viewContents =
   FileManage::document->
   subseqence(ScrollBar::handlePosition,
   ScrollBar::handlePosition + ViewPort::height - 1)}

FileManager

+document : sequence(lines)

{context linesVisible
inv: ViewPort::viewContents->size()=
ViewPort::height.max(FileManager::document->size())}

6/18/2007   © 2007, Spencer Rugaber   10

OCL Postcondition Constraint

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

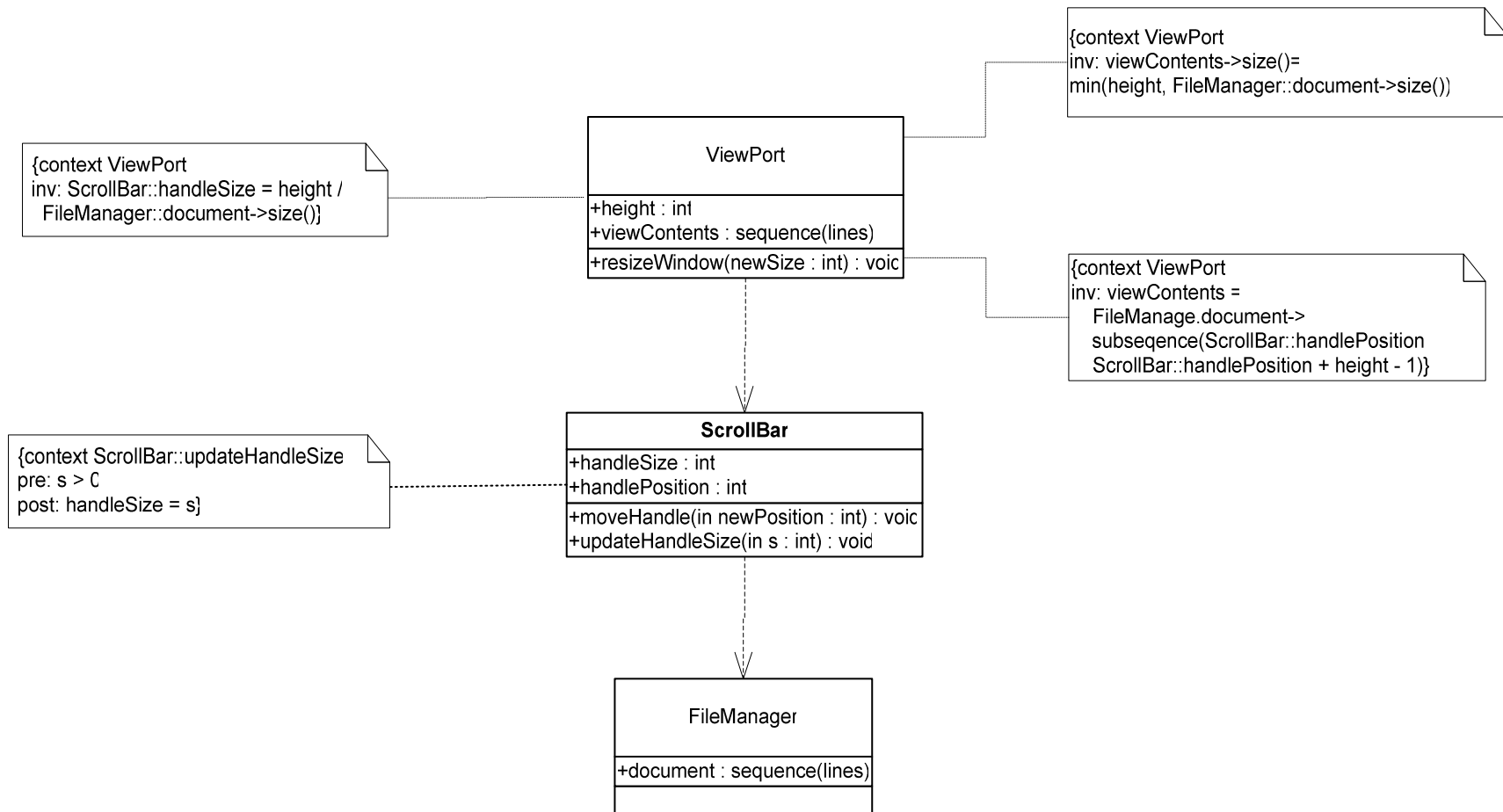| **ScrollBar** |
| --- |
| +handleSize : int
+handlePosition : int |
| +moveHandle(in newPosition : int) : void |
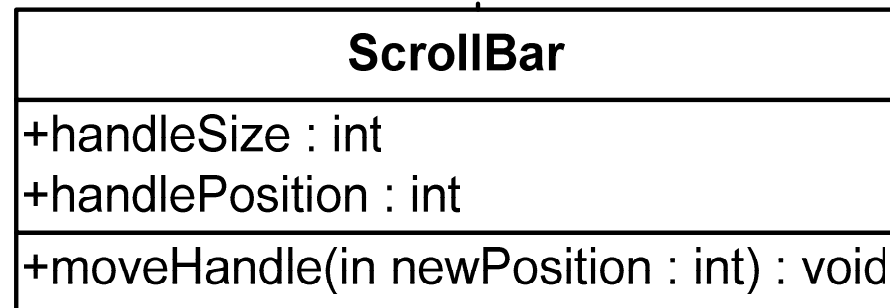
OCL Invariant Constraint

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

Phase 2

- Choose an architectural style
  - For example, layered, implicit invocation
- Assign components to layers
  - Typically user events are at the bottom; percepts are at the top
- Determine dependencies
- Update OCL
  - Constructive/applicative format (single variable on the left hand side

Phase 2

{context ViewPort
inv: viewContents->size()=
min(height, FileManager::document->size())

ViewPort

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

+height : int
+viewContents : sequence(lines)

+resizeWindow(newSize : int) : void

{context ViewPort
inv: viewContents =
  FileManage.document->
  subseqence(ScrollBar::handlePosition
  ScrollBar::handlePosition + height - 1)}

ScrollBar

{context ScrollBar::updateHandleSize
pre: s > 0
post: handleSize = s}

+handleSize : int
+handlePosition : int

+moveHandle(in newPosition : int) : void
+updateHandleSize(in s : int) : void

FileManager

+document : sequence(lines)

Updated OCL

- **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())

Layered, Implicit-Invocation Architecture

- *Layering:* Component composition in which lower-level components are unaware of how they are being used by upper-level components
  - Lower layers handle external events, propagating status changes upward
- Propagation is *implicit*--event announcement is made without the source component knowing the recipient; reduces coupling
  - Upper layers receive notifications, prepare and present results
- Benefits
  - Improved reusability: because lower-level components do not depend upon upper-level components
  - Reduced complexity: because of fewer allowed interactions among components
- Cost: Overhead due to the extra levels of indirection

Aside: Implementation Approach

- Status change initiated by external event
- Recipient component has assignments to its (status) variables overridden
- Overriding code notifies dependent components
- Code to do this generated automatically from OCL model
- C++ operator overload plus template mixin wrappers

Aside:
TextBrowser Product Family

- FileManager: source of data
  - **Static** or streaming
  - Supplies contents to Viewport for viewing

- ViewPort: information visualization
  - **Text** or statistics
  - Resizable display of file contents

- ScrollBar: controller
  - **Scrolling** or textField
  - Controls portion of file to be displayed

Aside: Alternative Invariant Maintenance Mechanisms

- Distributed
  - Each component knows about dependent components and invokes them when its state changes

- Aggregated
  - Single component responsible for handling all external events and delegating handling to subordinates

- Mediators
  - Special class whose instances are responsible for handling invariants; knows about independent and dependent participants

Invariant Maintenance Strategies

- The key design issue is deciding which class is responsible for ensuring the invariant

- As an example, consider the following situation. There are three objects: a file, a viewing window and a scrollbar. There is a constraint that says that the viewing window displays the part of the file contents that occurs at the position in the file that corresponds to the position of the scrollbar handle in the scrollbar tray

Example Continued

- If the user moves the scrollbar handle, the invariant is temporarily broken, because the displayed lines no longer represent those that exist at the requested portion of the file

- The next four slides give four strategies for reestablishing the invariant

AGGREGATION

- One of the objects (say the viewing window) owns (has as pointers or instance variables) the other two
- The scrollbar change request first comes to the owning instance (the viewing window) and gets delegated to the scrollbar, which returns a new position
- The viewing window then determines that it needs additional content in order to satisfy its responsibilities
- It makes a request to the file for the required lines and then displays them
- That is, the viewing window had aggregated the responsibility for the invariant maintenance

Distributed

- The scrollbar receives the change request and determines the new value (relative position in the scrollbar tray)

- It also knows that the viewing window depends on this information, so it makes a method call, passing the relative position

- The viewing window compares the relative position it received to the current value associated with the top displayed line and realizes that it cannot satisfy its responsibility

- It formulates a request to the file for the additional lines and sends a message to the file object

- The file object returns the lines to the viewing window for display

- That is, knowledge of the invariant is distributed among three objects that delegate partial responsibility to each other

Mediators

- A new object is introduced of class `Mediator`

- Each instance of `Mediator` is responsible for one constraint and know of its dependent objects

- The dependent objects know only that they must inform the `Mediator` when their attributes change value

- When the scrollbar is adjusted, it alerts the `Mediator`, which, in turn, requests the new position from the scrollbar

- The `Mediator` realizes that new content is required from the file, requests it, and passes it to the viewing window

- That is, the `Mediator` has knowledge and responsibility for invariant maintenance. The `Mediator` is an example of a design pattern

Mode Components
(Stirewalt and Rugaber)

- Attributes mentioned in invariants are called *status variables* and are implemented in such a way that when they change, any other dependent objects are transparently notified
  - C++ assignment override
- Invariant update code is compiled into a wrapper on the dependent object
  - Neither the independent nor dependent object is aware of the update (as far as code changes go). This is sometimes called *implicit invocation*
- That is, invariant update is handled by specially compiling the OCL constraint
- Wrappers and listeners/observers (implementation of implicit invocation) are further examples of design patterns

Using UML for Architectural Modeling

- Motivations
  - Accessibility
  - Use of UML tools
  - Compatibility with design models
  - Standardization

Future Work
# DYNAMO Interpretation of UML

UML Concept	Interpretation
Model	*Assembly*
Package	*Layer*
Class	*Component*
Attribute	*Percept*
Association	*Invariant*
Dependency	*Event*

- Suggests defining UML profile (stereotypes and meta-model constraints)

Requirements

- Focus on structural concerns
- Stylistic issues
- Behavioral
- Component interaction paradigms
- Constraints

UML MetaModel (Partial)
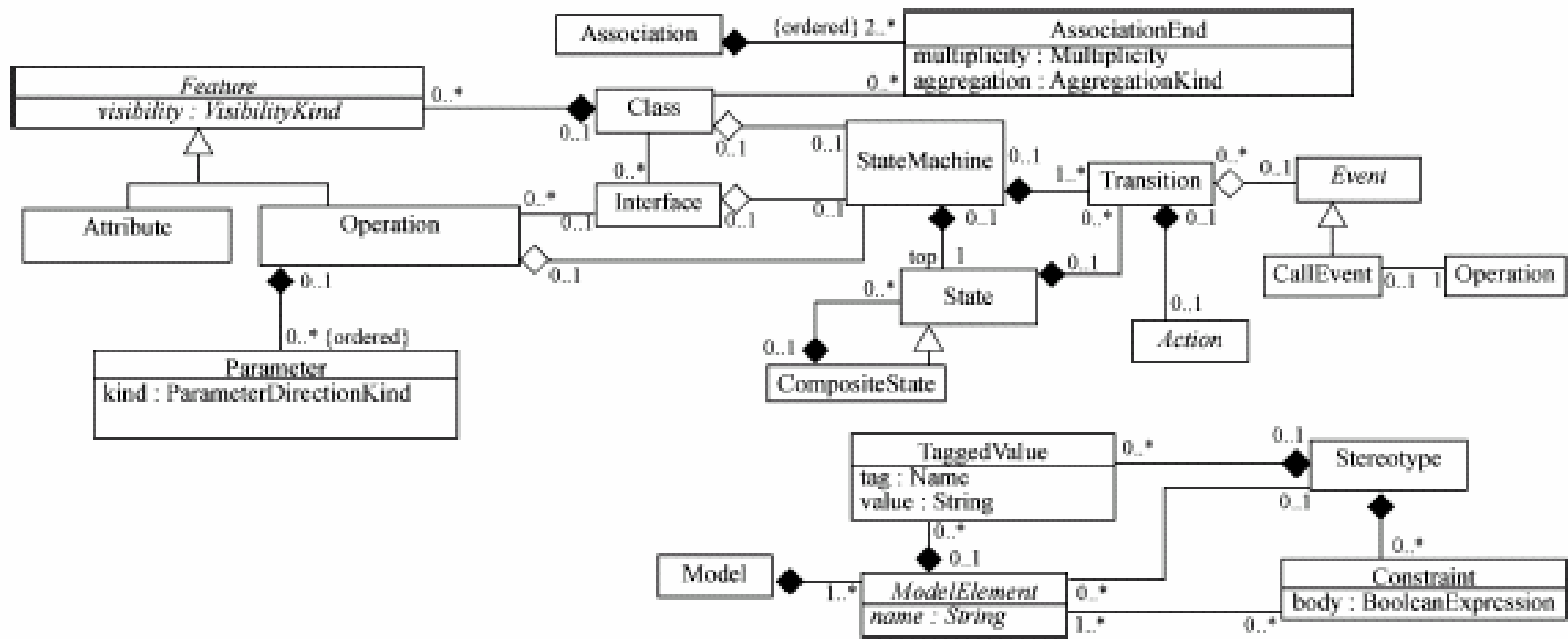
Assumption

- Tools exist to check the application of OCL constraints to UML models

- Includes ability to annotate and check the UML Meta Model

- Poseidon (http://gentleware.com/index.php)

- Octopus (http://www.klasse.nl/octopus/index.html)

Approach

- Determine modeling vocabulary
- Define a new stereotype for each term
- Compose OCL metamodel constraints to express participation of vocabulary in models
  - Only properly stereotyped elements can participate
  - Counts must be correct
  - Type checking rules
  - Express style rules as appropriate
    - e.g. directionality of data or control flows
- Consider adding tagged values to annotate new model type {profile}