Automated Invariant Maintenance via OCL Compilation

Spencer Rugaber

College of Computing

Georgia Institute of Technology

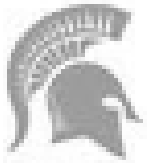
DYNAMO project sponsored by DARPA

Other Participants

Kurt Stirewalt

Computer Science and Engineering

Michigan State University



Georgia Tech students

Jonathan Gdalevich, Corinne McNeely, Terry Shikano, Patrick Yaner, David Zook

Michigan State students

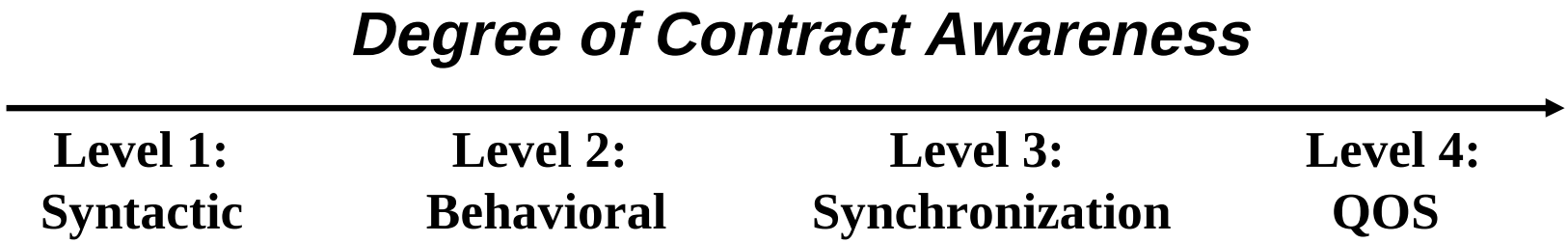
Reimer Behrends, Scott Fleming, AliReza Namvar

Component Assembly Problem

- Given:
  - Set of reliable and well-understood components
  - Specification of desired assembly properties (guarantees)
- Goal:
  - Provide a reliable assembly of the components satisfying the guarantees and the following non-functional requirements
    - Intentionality - enforcement of guarantees manifest
    - Transparency - no intrusion into components
    - Maintainability - abstract, model-based specification
    - Flexibility - support various collaboration mechanisms
    - Economy - no additional overhead vs. traditional composition
- Domain:
  - Interactive systems with user-initiated events that can change system status and resultant presentations

Guarantees

- Beugnard *et al.,* IEEE Computer'99



- Higher degrees of contract awareness:
  - Provide more power but
  - Require more infrastructure to implement

DYNAMO Approach

1. Model-based specification (UML and OCL)
2. Three-phased design method
3. **Efficient implementation of guaranteed behavior via metaprogramming**
4. Tool support

1. Model-Based Specification

- Use mathematical formalism to express system properties (*what*) while avoiding implementation details (*how*)
- Approaches
  - Assertional/axiomatic: pre/post conditions
  - Algebraic: equations interpreted as rewrite rules
  - Model-based: state modeling + assertions

Advantages of Formal Specification

- High-level; avoids implementation details
- Facilitates maintenance by reducing total amount of written code
- Enables reasoning about correctness
- Supports automatic code generation to increase productivity

Example
TextBrowser Product Family

- ViewPort: information visualization
  - **Text** or statistics
  - Resizable display of file contents
- ScrollBar: controller
  - **Scrolling** or textField
  - Controls portion of file to be displayed
- FileManager: source of data
  - **Static** or streaming
  - Supplies contents to Viewport for viewing

UML

- Unified Modeling Language
- Industry standard (OMG)
- Nine (13) diagram types + OCL
- Booch, Jacobson, Rumbaugh
- Rational Corp. (now IBM)
- CASE-tool support (Rational Rose, ArgoUML)

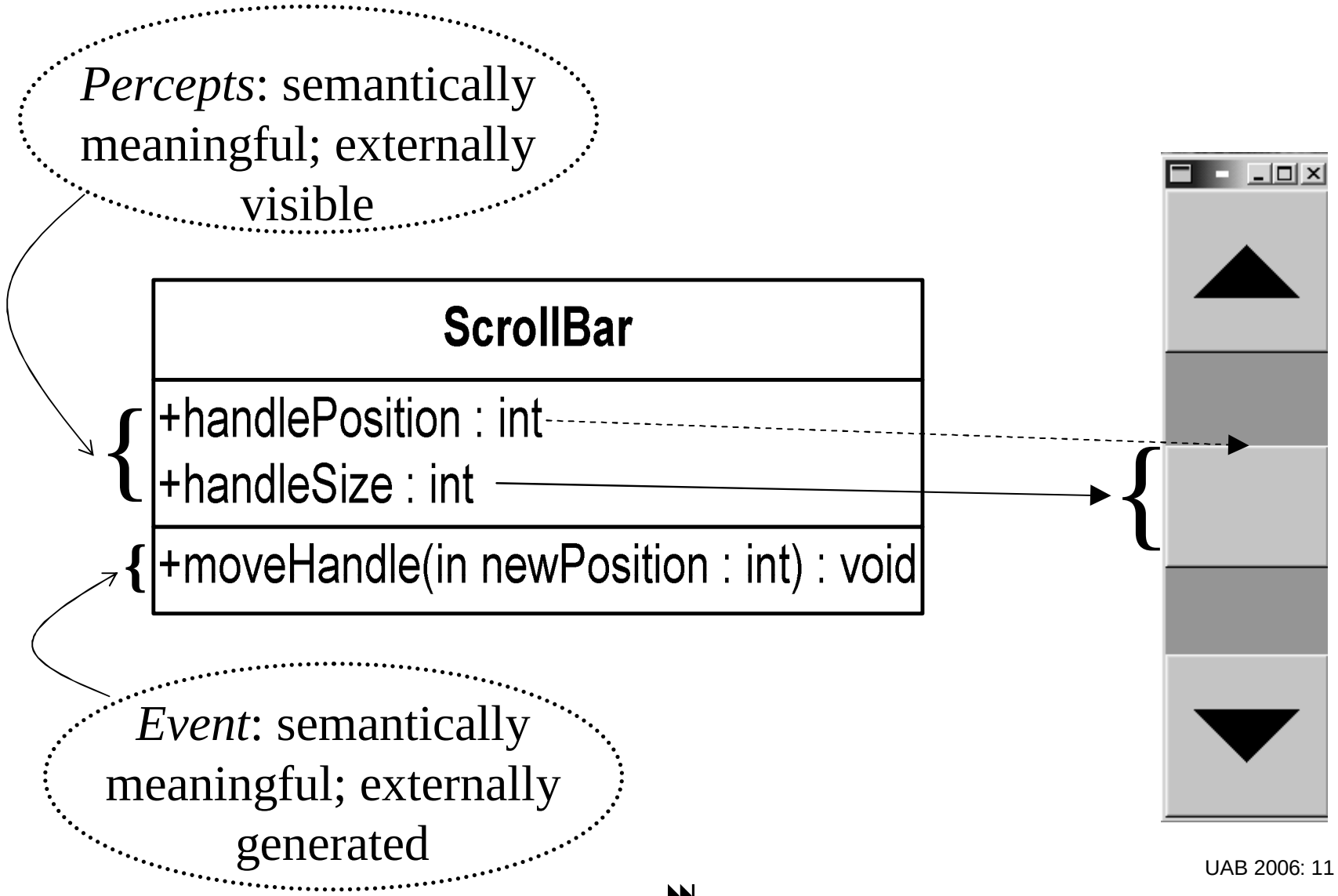
⏭

DYNAMO Interpretation of UML

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

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

Single Component



Assembly of Components

Displays

ViewPort

height : int
contents : sequence(String)

resize(in newSize : int) : void

Lines Visible

FileManager

document : sequence(String)

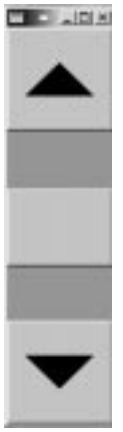
ScrollBar

+handleSize
+handlePosition

+moveHandle(in newPosition : int) : void

HandleProportion

Once upon a midnight dreary,
while I pondered weak and weary,
Over many a quaint and curious
volume of forgotten lore—
While I nodded, nearly napping,
suddenly there came a tapping,
As of someone gently rapping,
rapping at my chamber door.
""Tis some visitor," I muttered,
"tapping at my chamber door—
Only this and nothing more."

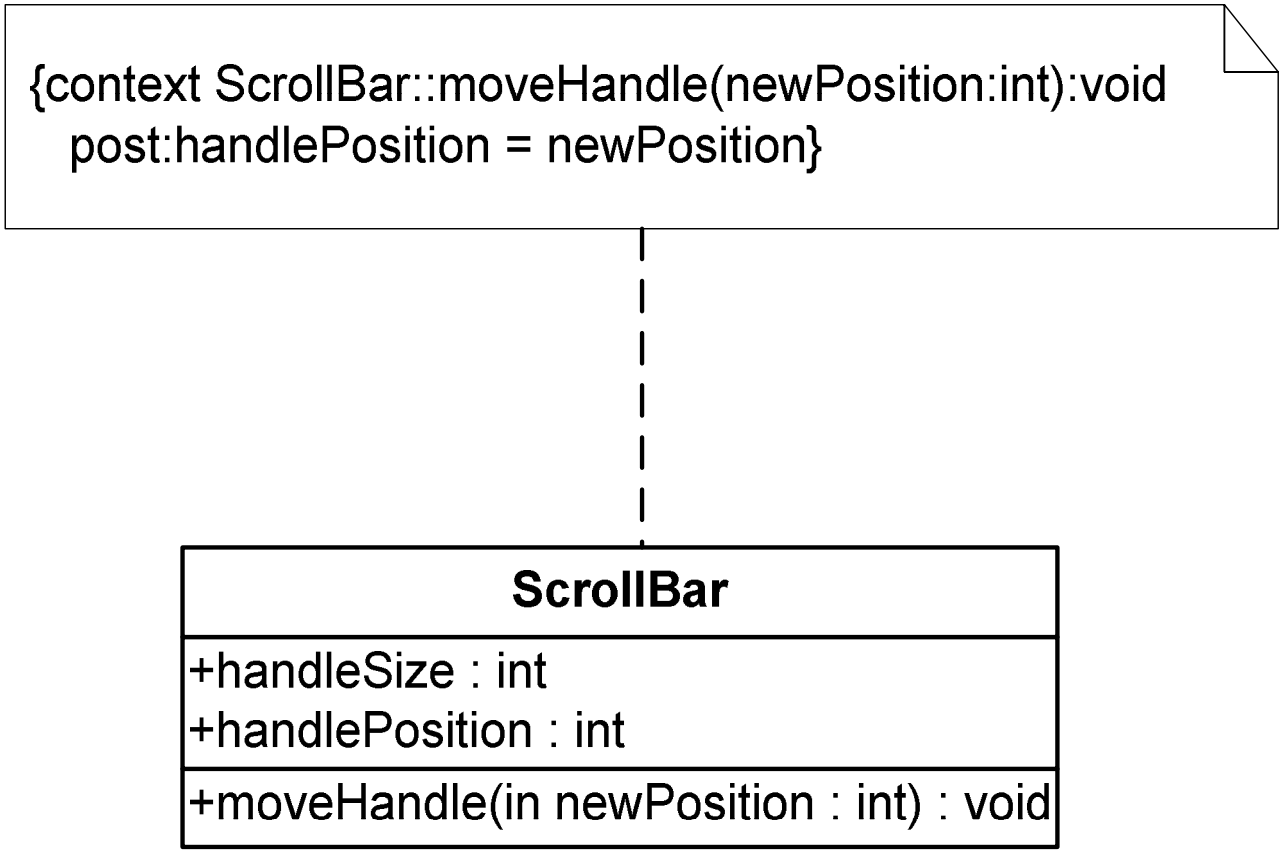


⏭

OCL

- Object Constraint Language
- UML class diagrams give a structural description but provide no semantics
- OCL adds constraints in the form of *invariants* and *pre/post conditions*
- OCL comprises first-order predicate logic, diagram navigation and collection classes

OCL Postcondition Constraint



⏭

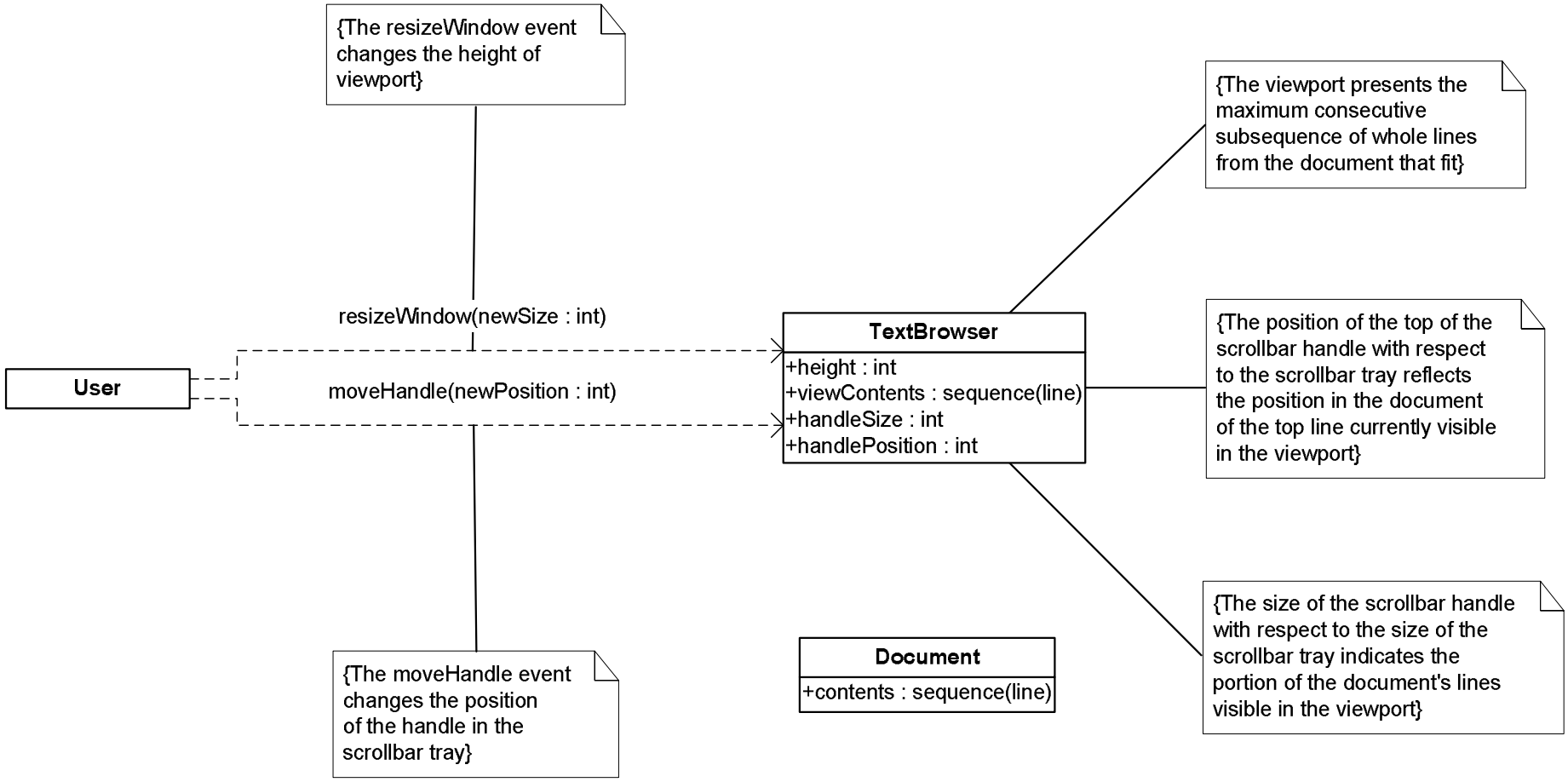
OCL Invariant Constraint

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

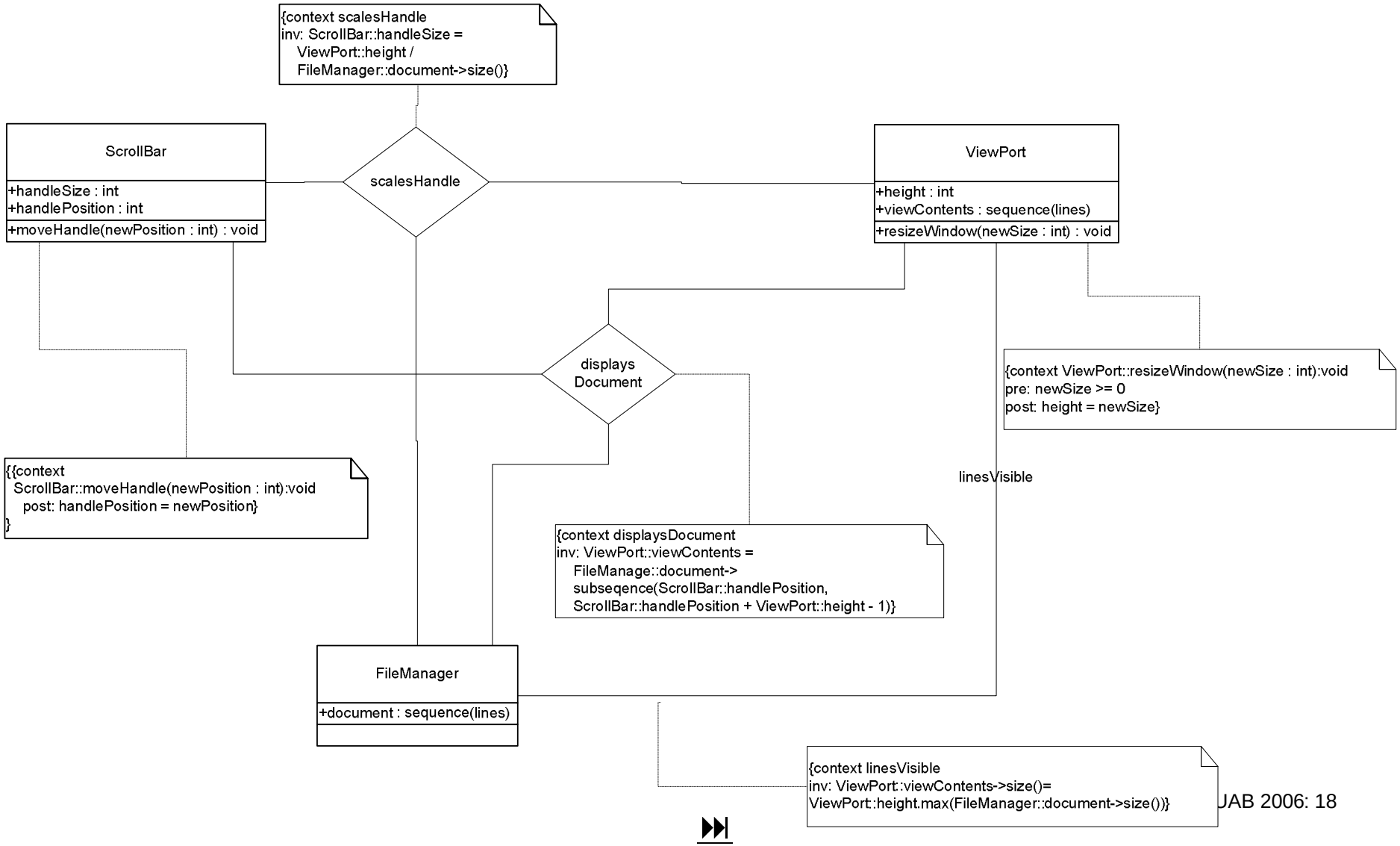
2. Three-Phase Design

0. **Context**: determination of interface between system and its environment: external actors; events, percepts, constraints expressed with natural language annotations
1. **Component**: decomposition of system into components, allocation of responsibilities; specification of OCL invariants and pre/post conditions
2. **Architecture**: layered, implicit invocation architecture, applicative constraints

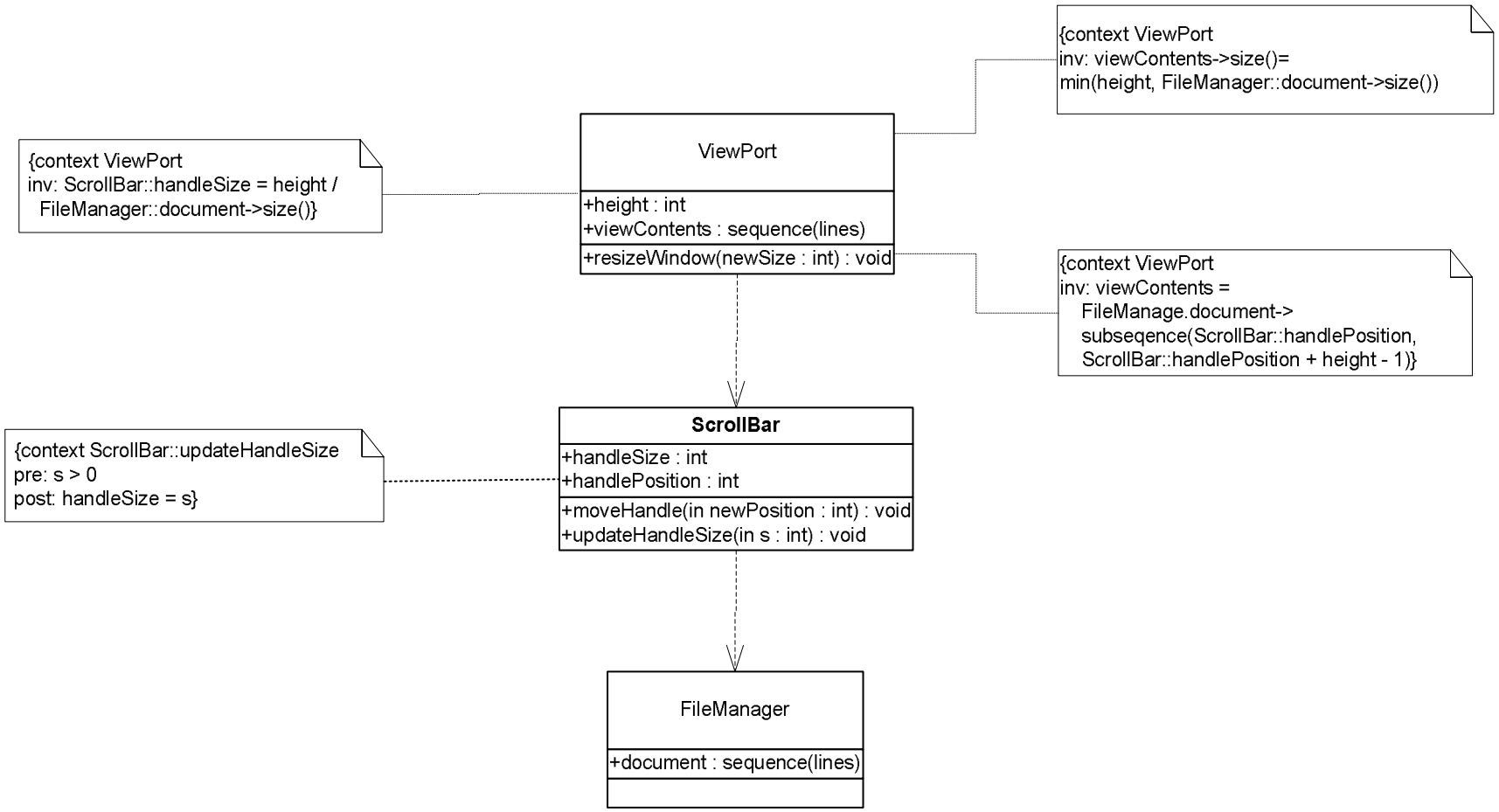
Phase 0


{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)
User
moveHandle(newPosition : int)
TextBrowser
+height : int
+viewContents : sequence(line)
+handleSize : int
+handlePosition : int
{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 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())}

Phase 2


{context ViewPort
inv: viewContents->size()=
min(height, FileManager::document->size())
{context ViewPort
inv: ScrollBar::handleSize = height /
FileManager::document->size()}
ViewPort
+height : int
+viewContents : sequence(lines)
+resizeWindow(newSize : int) : void
{context ViewPort
inv: viewContents =
FileManage.document->
subseqence(ScrollBar::handlePosition,
ScrollBar::handlePosition + height - 1)}
{context ScrollBar::updateHandleSize
pre: s > 0
post: handleSize = s}
ScrollBar
+handleSize : int
+handlePosition : int
+moveHandle(in newPosition : int) : void
+updateHandleSize(in s : int) : void
FileManager
+document : sequence(lines)

3. Efficient Implementation of Guaranteed Behavior

- **Problem:** assembling components in such a way that information hiding is upheld and guarantees are enforced without compromising efficiency. This is called the *invariant maintenance* problem
- **Solution:** Status Variables + Mode Components
  - Layered, implicit invocation architecture: non-invasive detection and notification of status change
  - Metaprogramming: Use of C++ compile-time features to avoid run-time overhead

Invariant Maintenance

- Invariants denote relationships among the externally visible status elements of the components of an assembly
- An event is detected by an appropriate component and handled, possibly altering that component's status
- The remainder of system components must react to reestablish invariants
- The goal is to do so in such a way that non-functional properties obtain

⏭

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

DYNAMO Layering

(GenVoca / Mixin layers: Batory & Smaragdakis)

- One or more components per layer
- Layering specified during Phase 2
- Layers implemented via mixin inheritance (superclass as template parameter)

⏭

DYNAMO Implicit Invocation

- 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 model

Metaprogramming

- *"The writing of programs that write or manipulate other programs (or themselves) as their data or that do part of the work that is otherwise done at runtime during compile time."* (Wikipedia)
- C++ template facility goes well beyond that of Java 5 and Ada 95 by providing a full compile time program generation facility
- Specifically, DYNAMO uses template-based mixin inheritance and method inlining to wrap components with added functionality

⏭

C++ Template Classes

- A template class in C++ is a parameterized description of a class
- The parameter is typically a type
- For example, if a programmer wishes to develop a stack class, a template can be used whose parameter is the type of the element on the stack. Then stacks of integers or strings or structs can be built
- Because the instantiation of the template class with its parameter to create the specific stack class is done at compile time, this is an example of metaprogramming

Mixins

- In DYNAMO, C++ template classes are used to provide *mixins*
  - A mixin is a class from which some target class inherits in addition to its normal base class (superclass)
  - The class being mixed in is provided as the template parameter. That is, the template class itself inherits from its parameter!
- Mixins allow programmers to "mix in" cross-cutting capabilities
- In DYNAMO, two kinds of capabilities are mixed in
  - Detection of changes to component status
  - Constraint reestablishment code
- The component class itself is *wrapped* by the mixin mechanism

Traditional Intercomponent Communication

```
class Server {
  public:
    void service() { ... }
};

class Client {
  public:
    void function(Server& s) { ...s.service();... }
};

// main
Client cl;
Server se;
cl.function(se);
```


Mixin Intercomponent Communication

```
class Server {
  public:
    void service() { ... }
};

template<class SERVER>
  class Client : SERVER {
    public:
      void function() { ...SERVER::service();... }
};  // No pointer indirection; inlining may remove call

// main
Client<Server> cl;
cl.function();
```


Status Variables

- *Status:* attributes of a component to which other components are sensitive
- *Status variable:* lightweight wrapper on a status attribute that detects changes to its value and invokes a method to handle announcements to dependent components
- Detection via overloaded assignment
- Wrapping done by C++ template

Toy Example

- Two components
  - Dependent: A with StatusVariable a
  - Independent: B with StatusVariable b
- Invariant
  - a must always equal twice b
- OCL

```
{context A inv: a = 2 * B.b}
```

Schematic for an Independent Component

```
class B {
  protected:
    int b;
  public:
    ...
    void tweak(const int& x) {
      b = x;
    }
    ...
};
```

Status attribute

Simulates status-change event

Status Variable Responsibilities

1. Detect changes
2. Notify dependent components
3. Initialization
4. Data storage and retrieval

StatusVariable Template Class

```
( 1) template <typename T>
( 2) class StatusVariable {
( 3)   public:
( 4)     StatusVariable() {}
( 5)     StatusVariable(const T& t) : data(t) {}
( 6)     virtual T& operator= (const T& t) {data = t;}
( 7)     virtual operator T() {return data;}
( 8)   protected:
( 9)     T data;
(10) };
```

Template parameter is original type of variable

3. Initialization

1. Change detection

4b. Value retrieval via type conversion

4a. Storage for attribute

2. Notification deferred to derived class

Generated Status Change Announcement Mechanism

```
( 1) template <typename T>
( 2) class SV_B_b : public StatusVariable<T> {
( 3)   public:
( 4)     SV_B_b() {}
( 5)     SV_B_b(const T& x) : StatusVariable<T>(x) {}
( 6)     void setUpdater1(Updaters* sc1P) {
( 7)       updater1P = sc1P;
( 8)     }
( 9)     T& operator=(const T& d) {
(10)       StatusVariable<T>::operator=(d);
(11)       if (updater1P)
(12)         updater1P->update1();
(13)     }
(14)   protected:
(15)     Updaters* updater1P;
(16) };
```

Naming convention: component B; status variable b

Binding to update method

Extension to base class, method

2. Notification

Status Variable Declaration

```
SV_B_b<int> b;
```

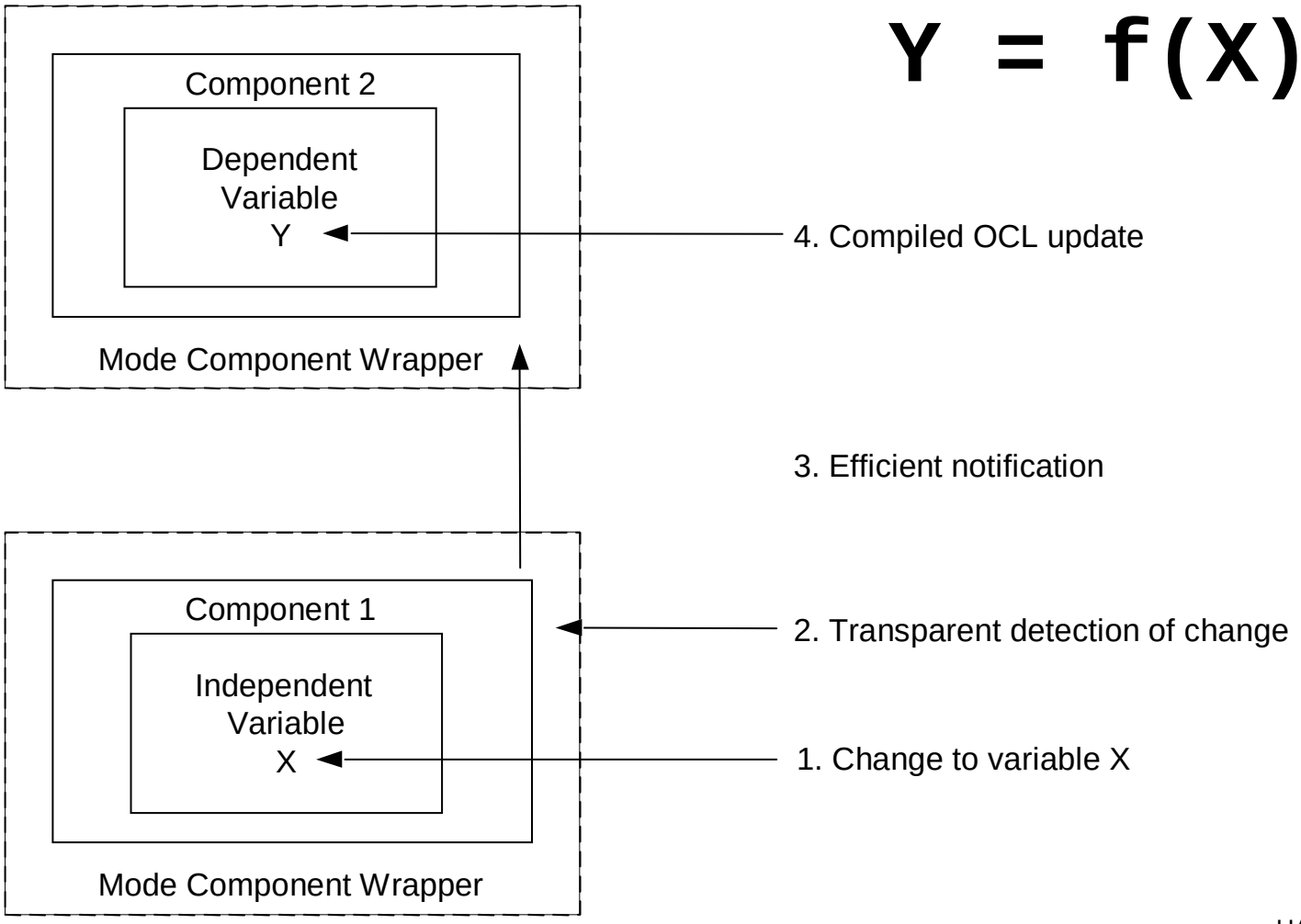

Status Variable Summary

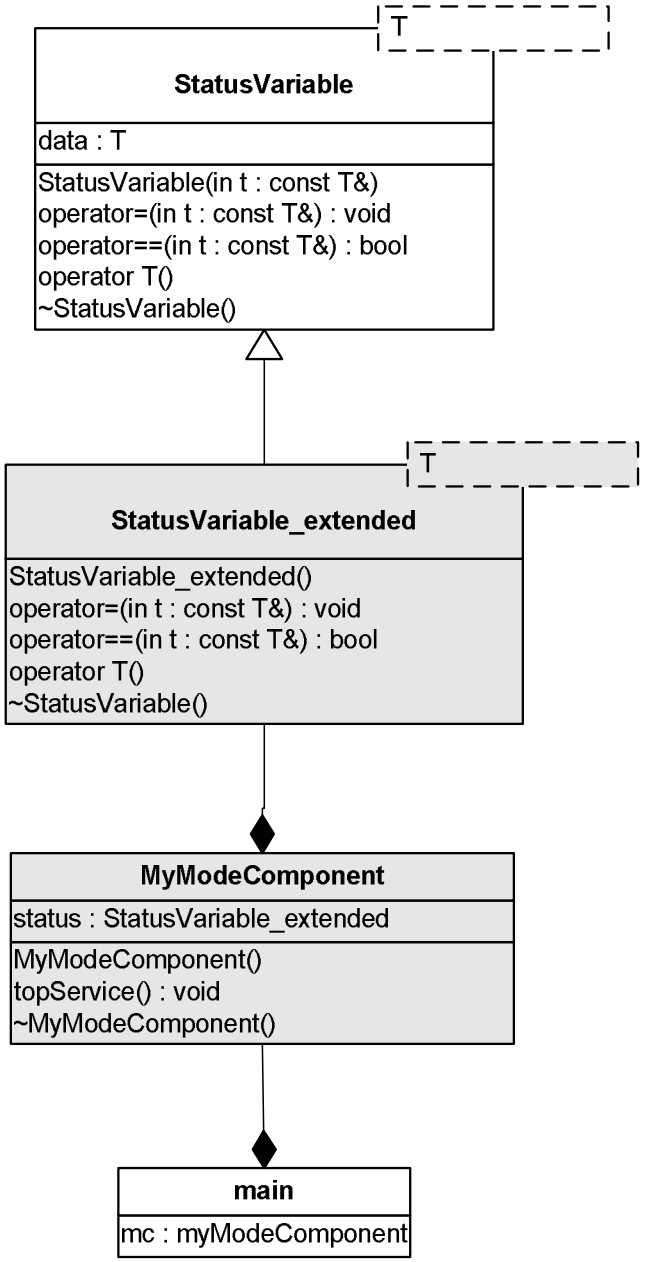
- For each component (B)
  - For each element of status in B (b of type T)
    - Generate a wrapper (SV_B_b) that
      - Detects and extends assignments to b
      - For each constraint in which b appears on the rhs
        - » Contains a pointer to an update method for the variable on the lhs of the constraint (`Updater1P`)
        - » Contains a method initializing the updater pointer (`setUpdater1`)
        - » Includes in the assignment overload method an indirect call to the updater method through the pointer (`operator=`)
      - Casts back to type `T` on retrieval (`operator T()`)

Mode Components

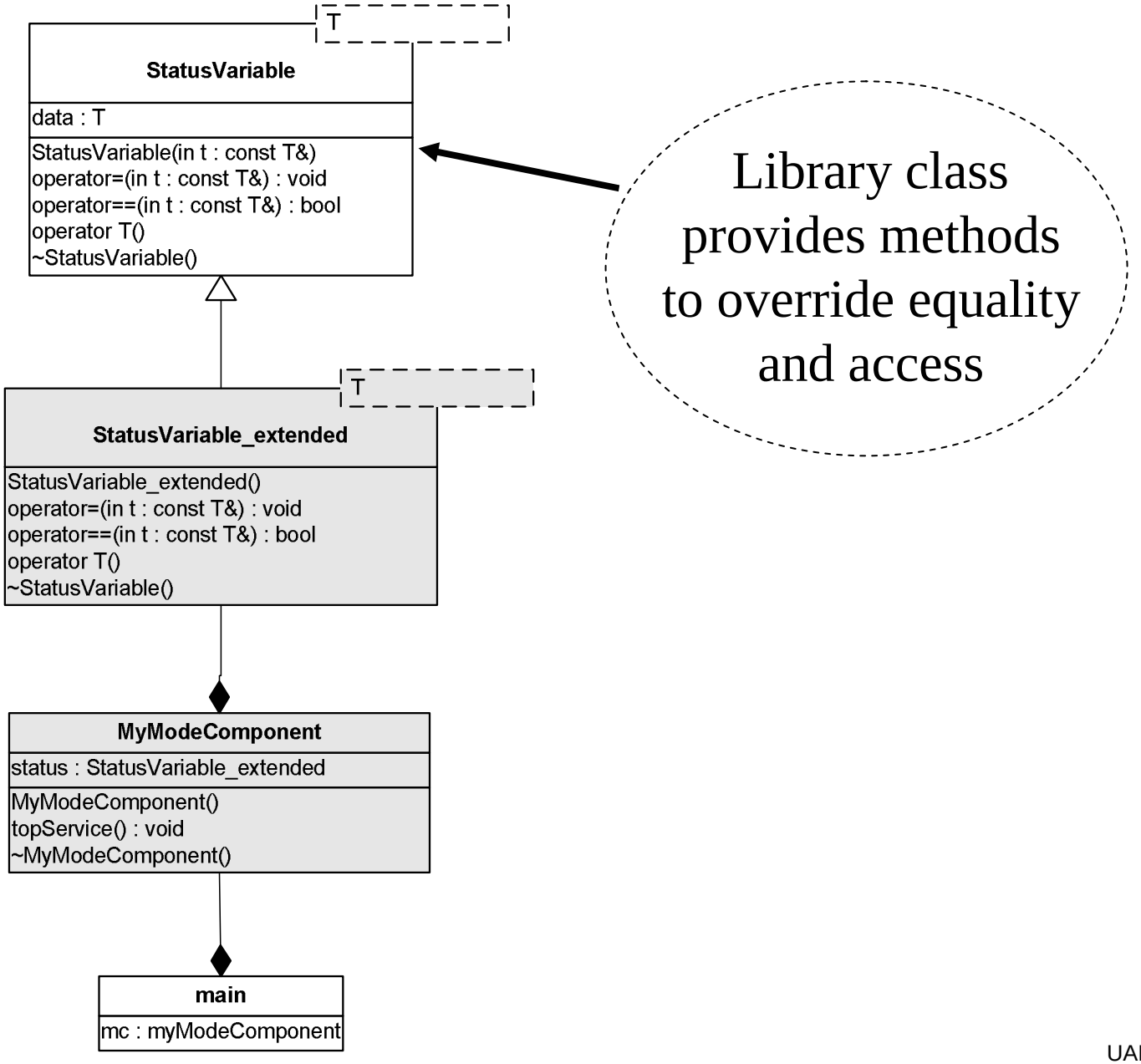
- Design idiom supporting invariant maintenance
- A *mode component* is a software component that make updates to its status (mode) available implicitly to client components
  - Mode components can also respond to service request made using traditional method calls
- Mode components are generated automatically by wrapping legacy components with templates
- There are two kinds of mode component wrappers
  - *Top wrappers* notify dependent components of status variable updates
  - *Bottom wrappers* receive notifications from independent components and update the appropriate contained status variable
- A mode component may have both kinds of wrapping

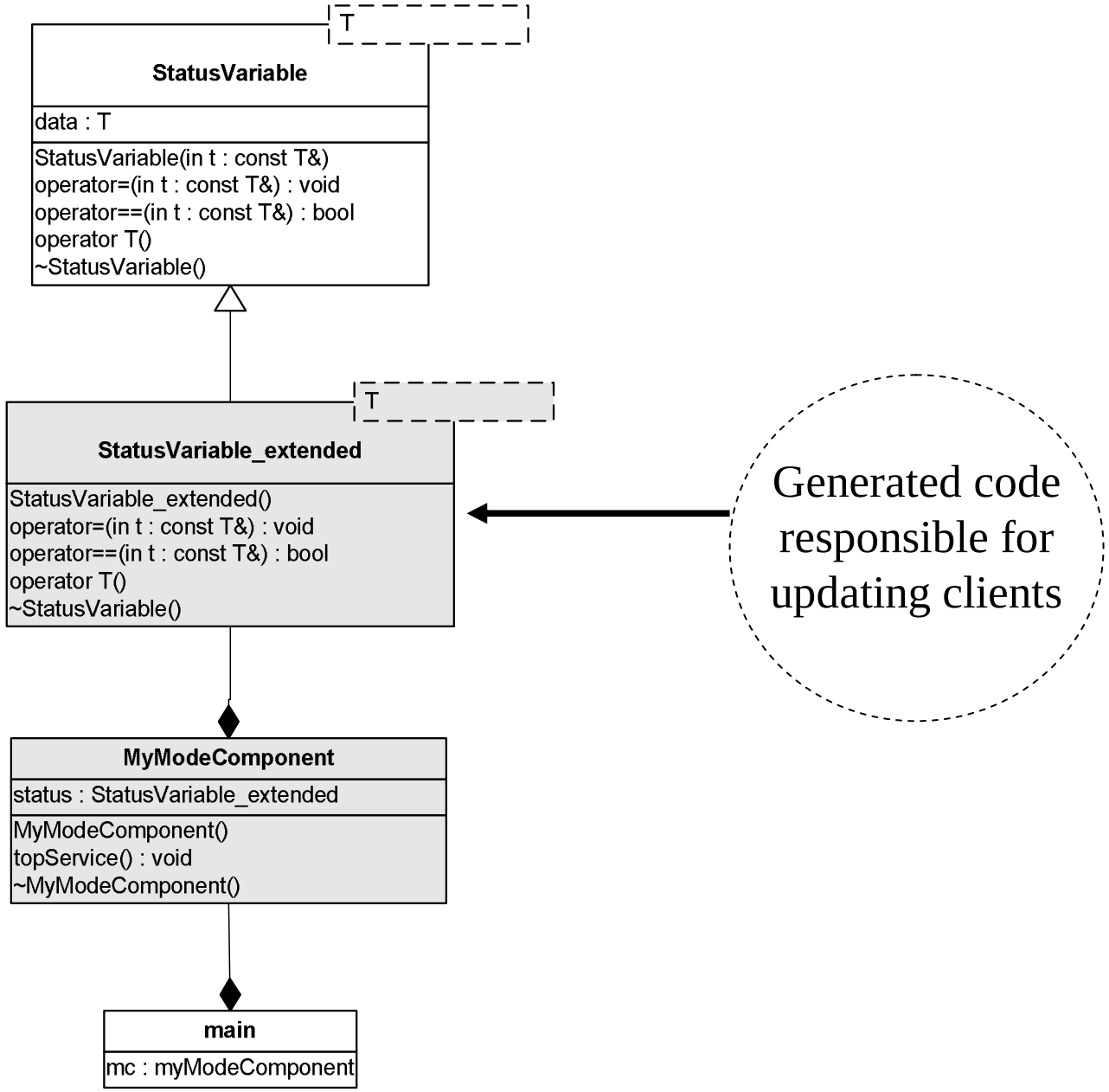
Mode Component Status Updates

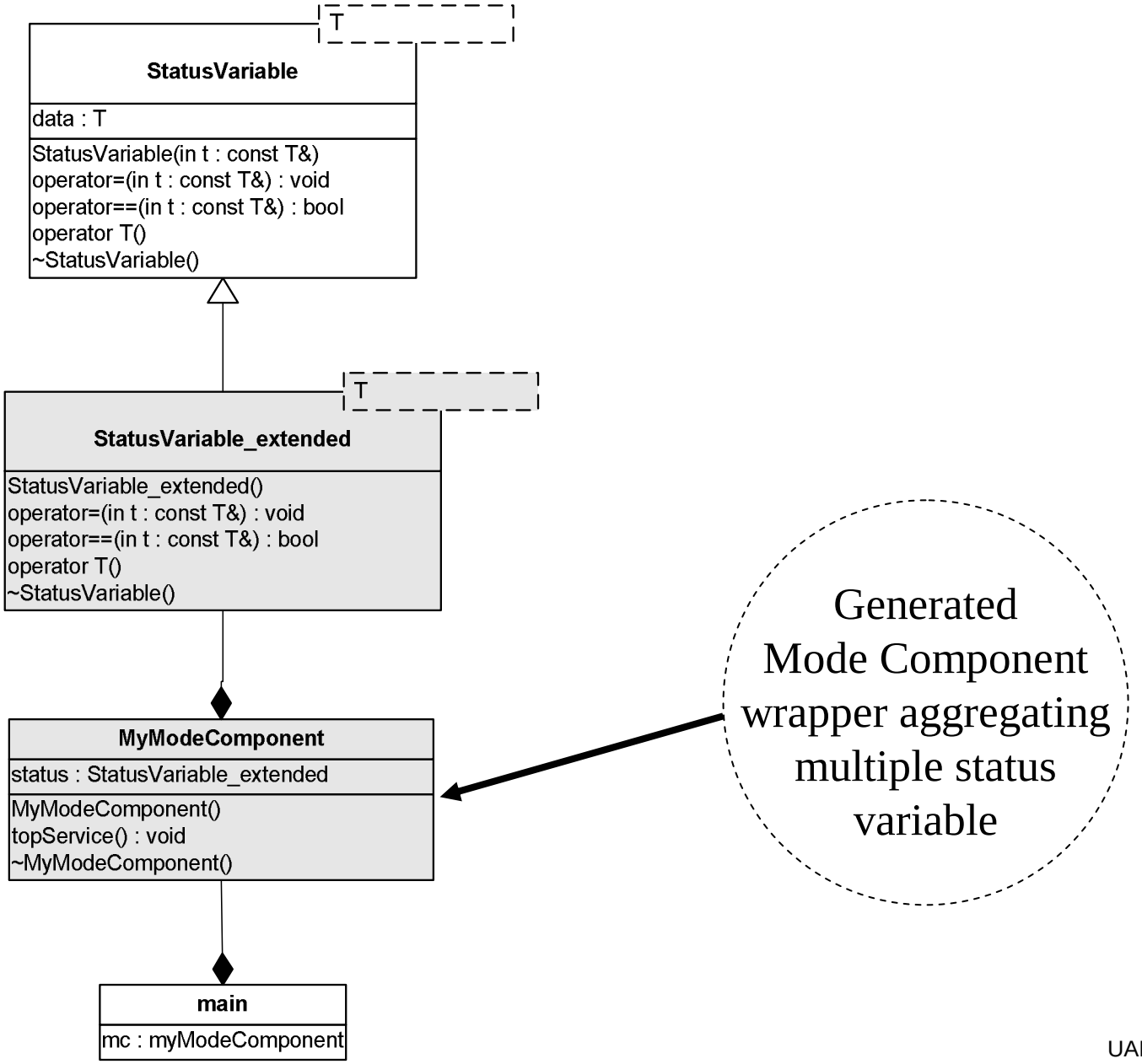


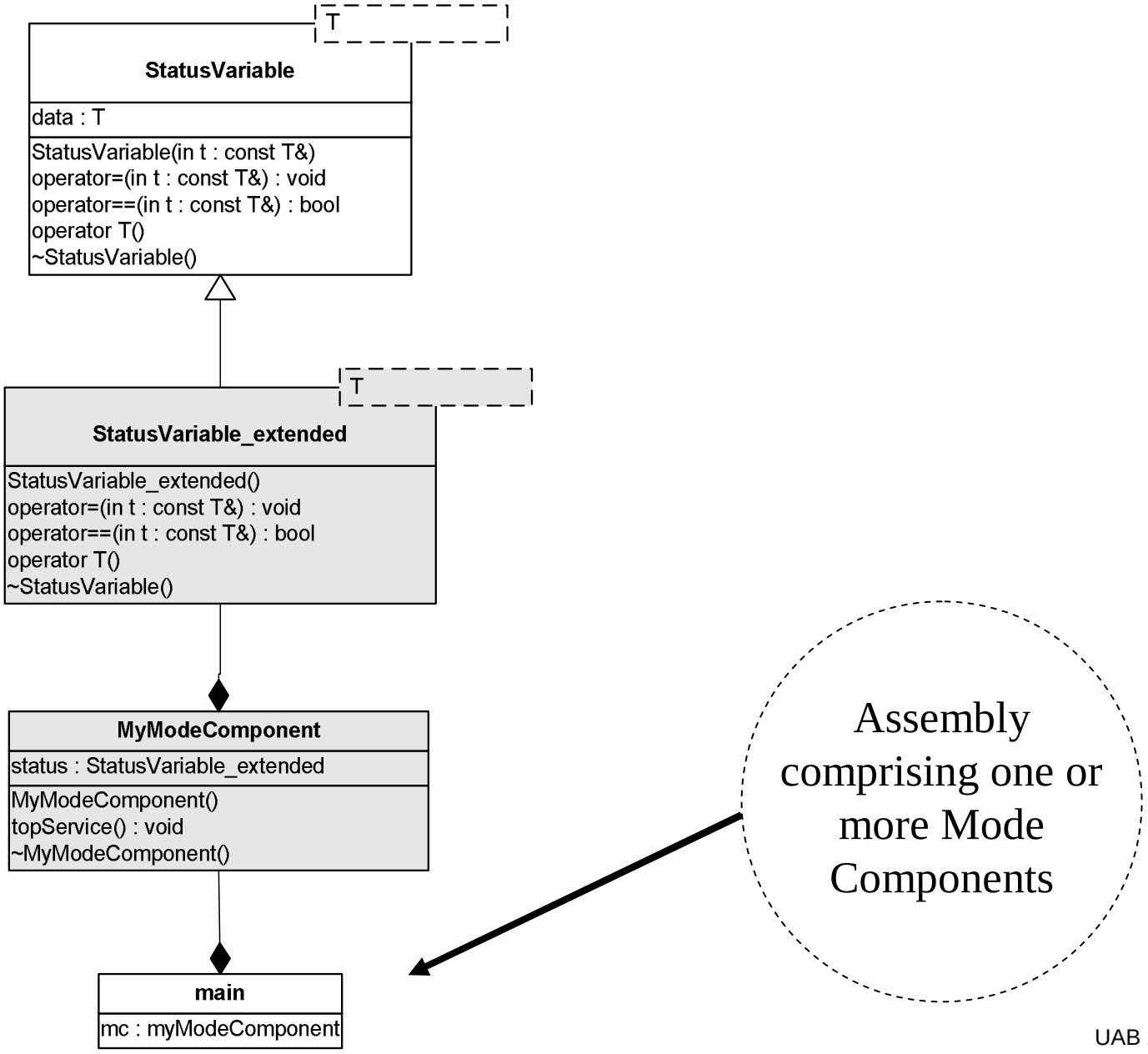


Mode Component Implementation


T
StatusVariable
data : T
StatusVariable(in t : const T&)
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
Library class provides methods to override equality and access
T
StatusVariable_extended
StatusVariable_extended()
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
MyModeComponent
status : StatusVariable_extended
MyModeComponent()
topService() : void
~MyModeComponent()
main
mc : myModeComponent


T
StatusVariable
data : T
StatusVariable(in t : const T&)
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
T
StatusVariable_extended
StatusVariable_extended()
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
Generated code responsible for updating clients
MyModeComponent
status : StatusVariable_extended
MyModeComponent()
topService() : void
~MyModeComponent()
main
mc : myModeComponent


T
StatusVariable
data : T
StatusVariable(in t : const T&)
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
T
StatusVariable_extended
StatusVariable_extended()
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
MyModeComponent
status : StatusVariable_extended
MyModeComponent()
topService() : void
~MyModeComponent()
main
mc : myModeComponent
Generated Mode Component wrapper aggregating multiple status variable

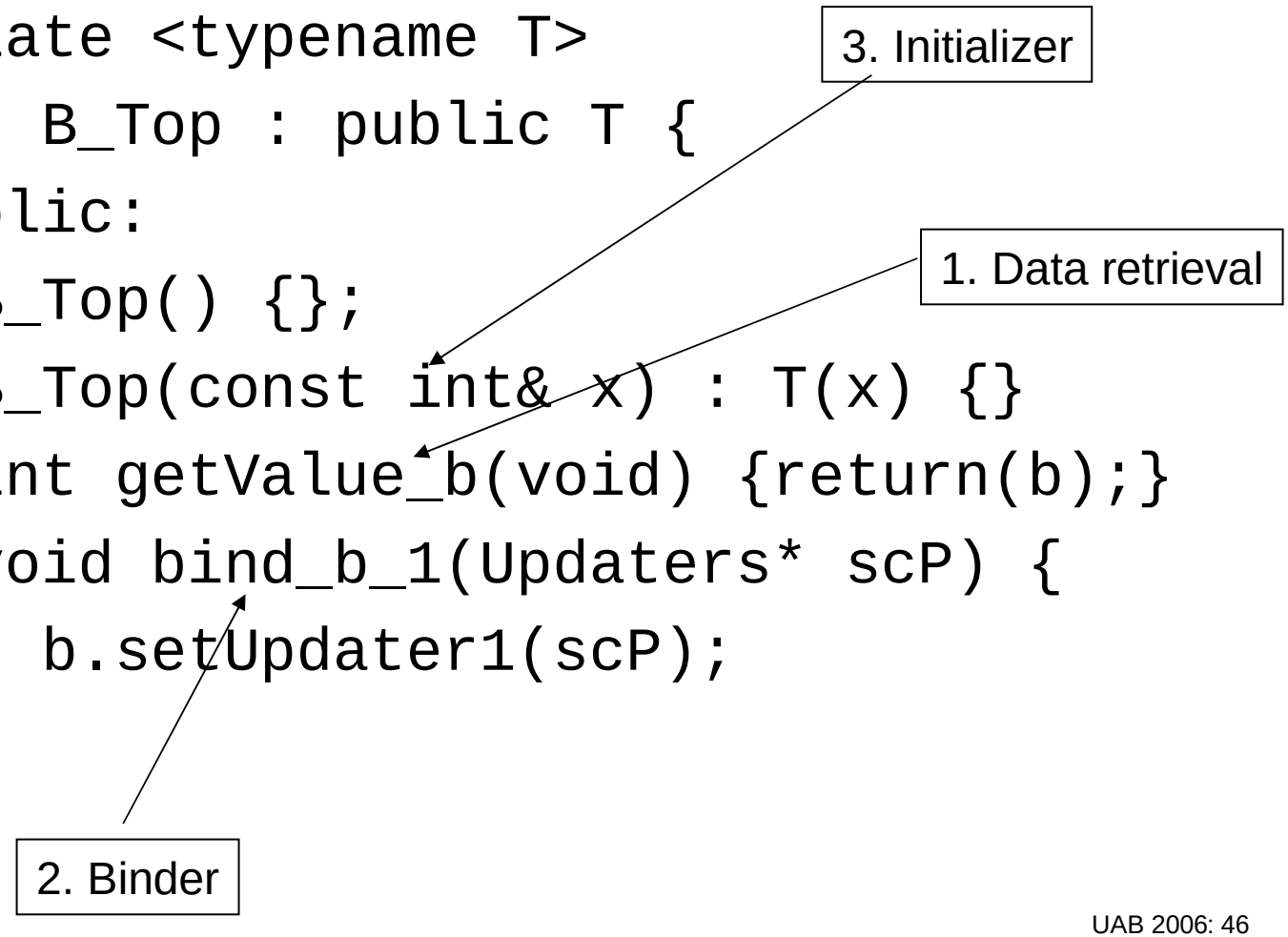

T
StatusVariable
data : T
StatusVariable(in t : const T&)
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
T
StatusVariable_extended
StatusVariable_extended()
operator=(in t : const T&) : void
operator==(in t : const T&) : bool
operator T()
~StatusVariable()
MyModeComponent
status : StatusVariable_extended
MyModeComponent()
topService() : void
~MyModeComponent()
main
mc : myModeComponent
Assembly comprising one or more Mode Components

Top Wrappers

- Top wrappers have the following responsibilities
  1. Provide access to status variable value
  2. Bind to dependent component
  3. Initialize status variable

Top Wrapper for Component B

```
( 1) template <typename T>
( 2) class B_Top : public T {
( 3)   public:
( 4)     B_Top() {};
( 5)     B_Top(const int& x) : T(x) {}
( 6)     int getValue_b(void) {return(b);}
( 7)     void bind_b_1(Updaters* scP) {
( 8)       b.setUpdater1(scP);
( 9)     }
(10) };
```



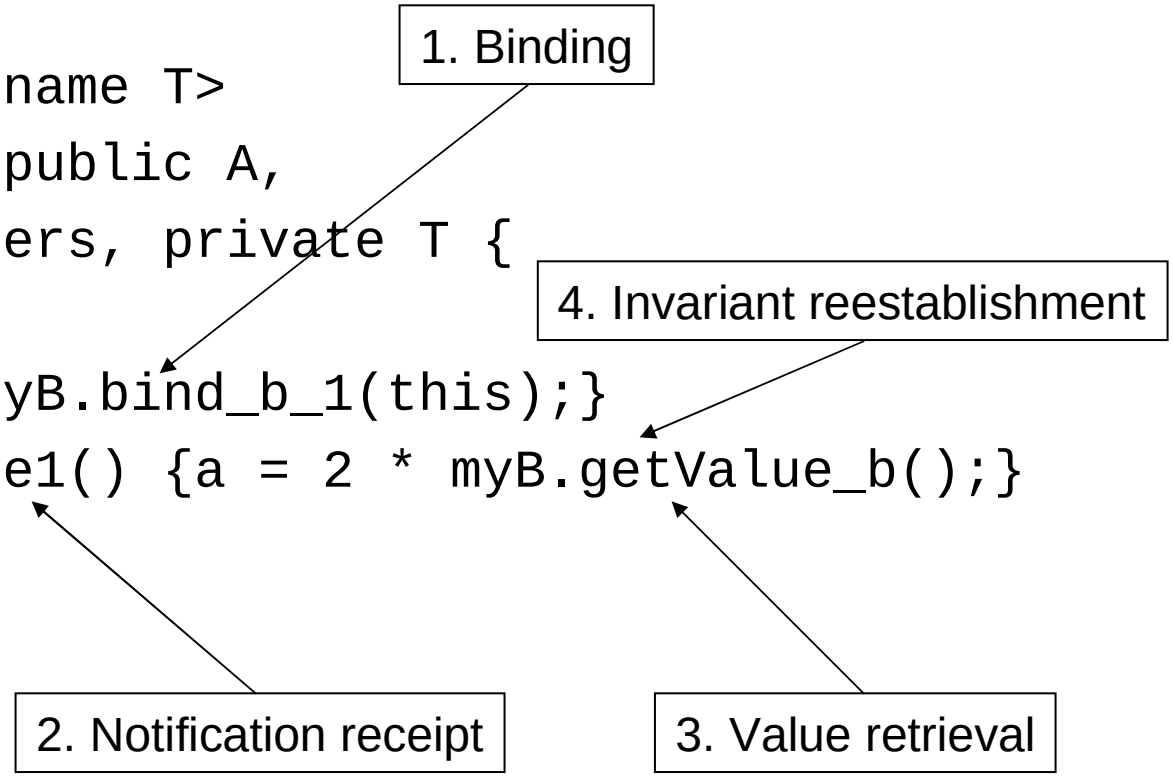
⏭

Bottom Wrappers

- Bottom wrappers have the following responsibilities
  1. Bind to independent component
  2. Receive notification of status changes
  3. Request new value
  4. Reestablish invariant

Bottom Wrapper for Component A

```
(1) template <typename T>
(2) class A_Bot : public A,
(3)   public Updaters, private T {
(4)   public :
(5)     A_Bot() {myB.bind_b_1(this);}
(6)     void update1() {a = 2 * myB.getValue_b();}
(7)   protected :
(8)     T myB;
(9) };
```



⏭

Other Features of Dependent Components

```
(1) template <typename T>
(2) class A_Bot : public A,
(3)   public Updaters, private T {
(4)   public :
(5)     A_Bot() {myB.bind_b_1(this);}
(6)     void update1() {a = 2 * myB.getValue_b();}
(7)   protected :
(8)     T myB;
(9) };
```

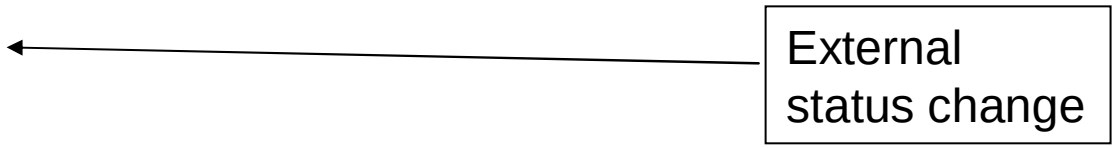
Wrapping

Private mixin

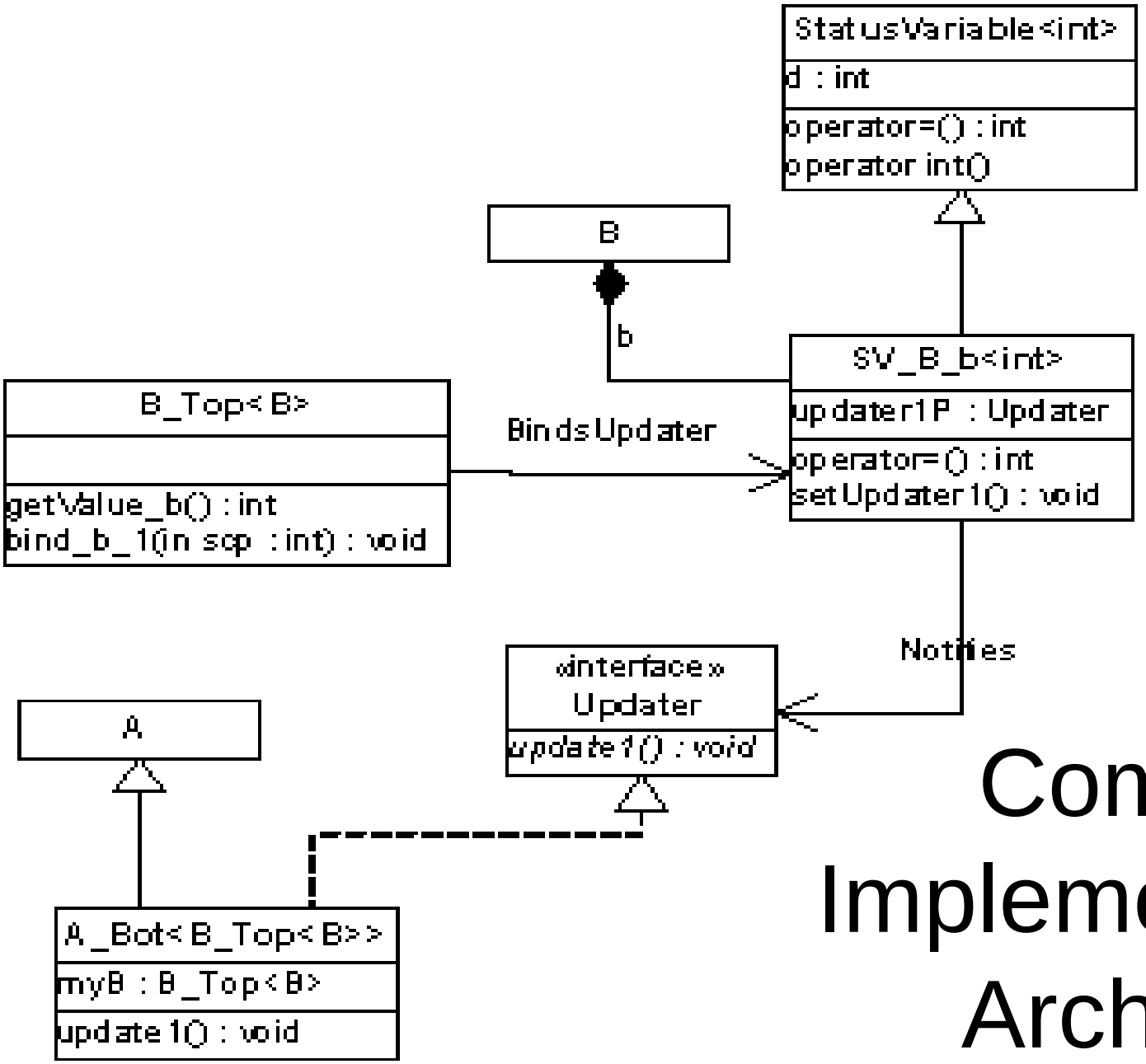
Potential inline

Aggregation of independent component

```
int main(void) {
  A_Bot<B_Top> myA(6);
  B_Top* lowerP = myA.getLowerP();

  cout << "b before change " <<
          lowerP->getValue_b() << endl;
  cout << "a before change = " <<
          myA.A::getValue() << endl;

  cout << "Change b to be 10" << endl;
  lowerP->tweak(10);

  cout << "b after change = " <<
          lowerP->getValue_b() << endl;
  cout << "a after change = " <<
          myA.A::getValue() << endl;
}
```



Use



Mode Component Implementation Architecture

⏭

Generalizations

- Combine top and bottom wrapping into one wrapper
- Wrap multiple status variables in the same component
- Support multiple constraints referring to the same status variable
- Allow multiple components per layer

Evaluation

- Intentionality
  - Visible implementation of each invariant
- Transparency
  - Only intrusion is declaration of status variable
- Maintainability
  - OCL + compilation
- Flexibility
  - Metaprogramming approach applicable to alternative update mechanisms
- Economy
  - Inlining and mixin inheritance avoids most indirection

Limitations

- Loss of symmetry
  - Due to template nesting
- Constructiveness
  - Not every constraint can be expressed in applicative form
- Circularities
  - Components may be mutually dependent
- Code obfuscation
  - Due to code generation and C++ template processing

Other Implementations of Invariant Maintenance

- Mode components are one solution to the problem of invariant maintenance. That is, how can remote parts of a system be kept up to date wrt changes in a given component
- A variety of alternative approaches have been suggested for how to maintain invariants
  - **Encapsulation/aggregation:** I.e. have one component aggregate the other and assume responsibility
  - **Distributed responsibility:** E.g. cascaded deletes)
  - **Separate component:** E.g. mediators, association classes

⏭

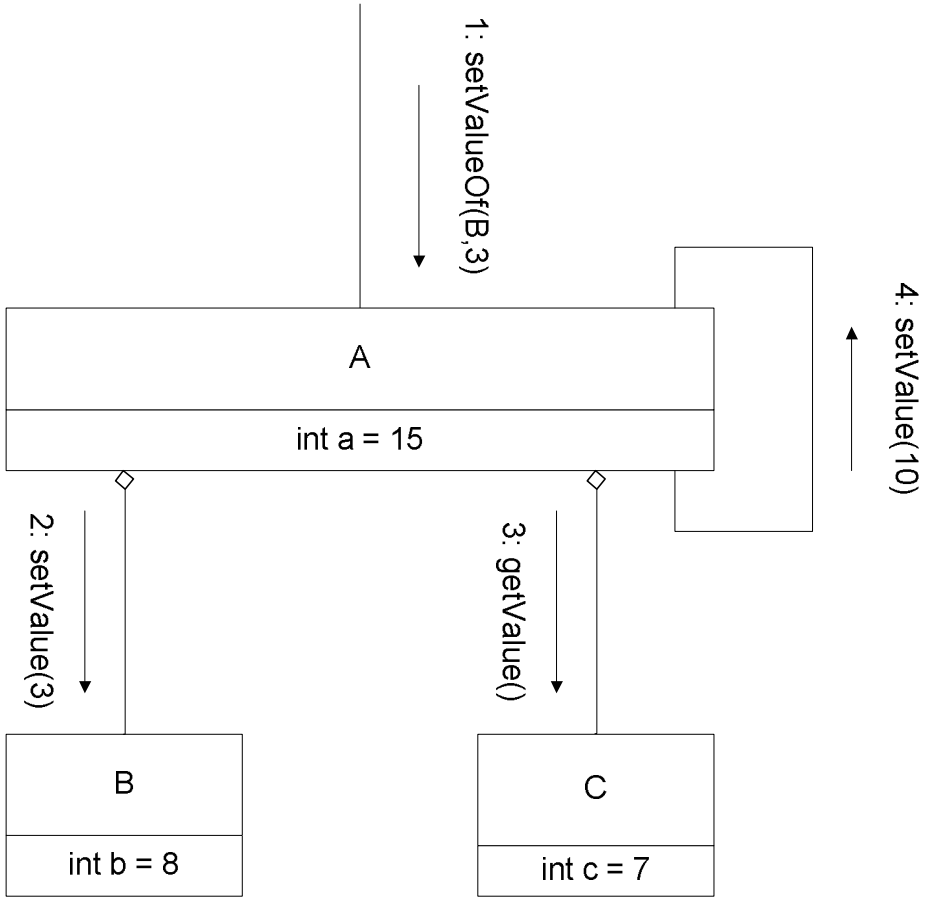
Matrix

	Ease of Manual Implementation	Transparency	Non Intrusiveness
Encapsulation	**High**	**Medium**	**Medium**
Distribution	**High**	**Low**	**Low**
Mediator	**Low**	**High**	**High**
Mode Component	**Low**	**High**	**High**

Comparative Scenario

- Three components (`A, B, C`) each holding a value (`a, b, c`), respectively
- Invariant among the components that requires that `a = b + c` hold in the face of changes to a or b
- Initial values `15, 8, 7,` respectively
- External component sends message updating the value of b to be 3

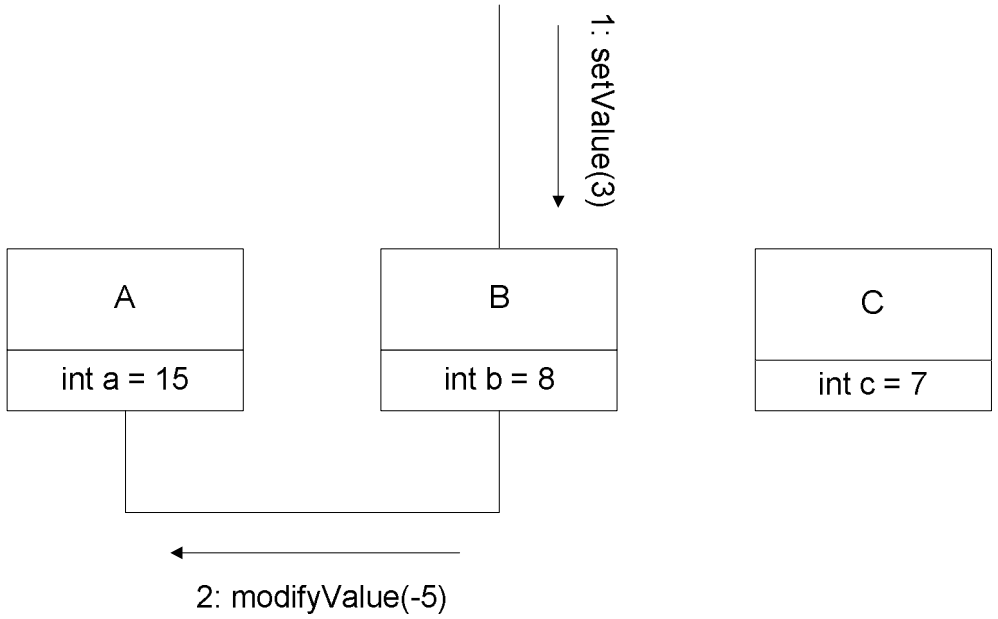
Encapsulation/Aggregation



- A encapsulates B and C from the rest of the components
- More method calls required
- B and C are available for black-box reuse

⏭

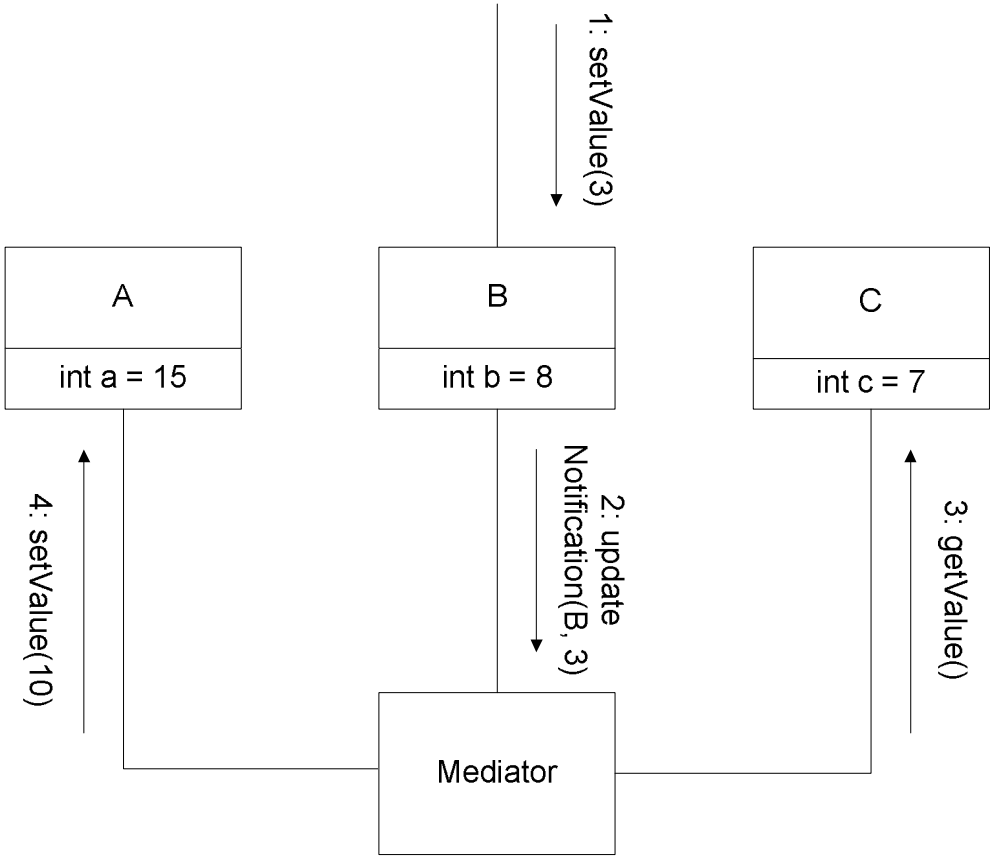
Distributed Responsibility / Collaboration



- Each component knows of its role in the collaboration
- Components can be used for white-box reuse
- Opportunities for optimization

⏭

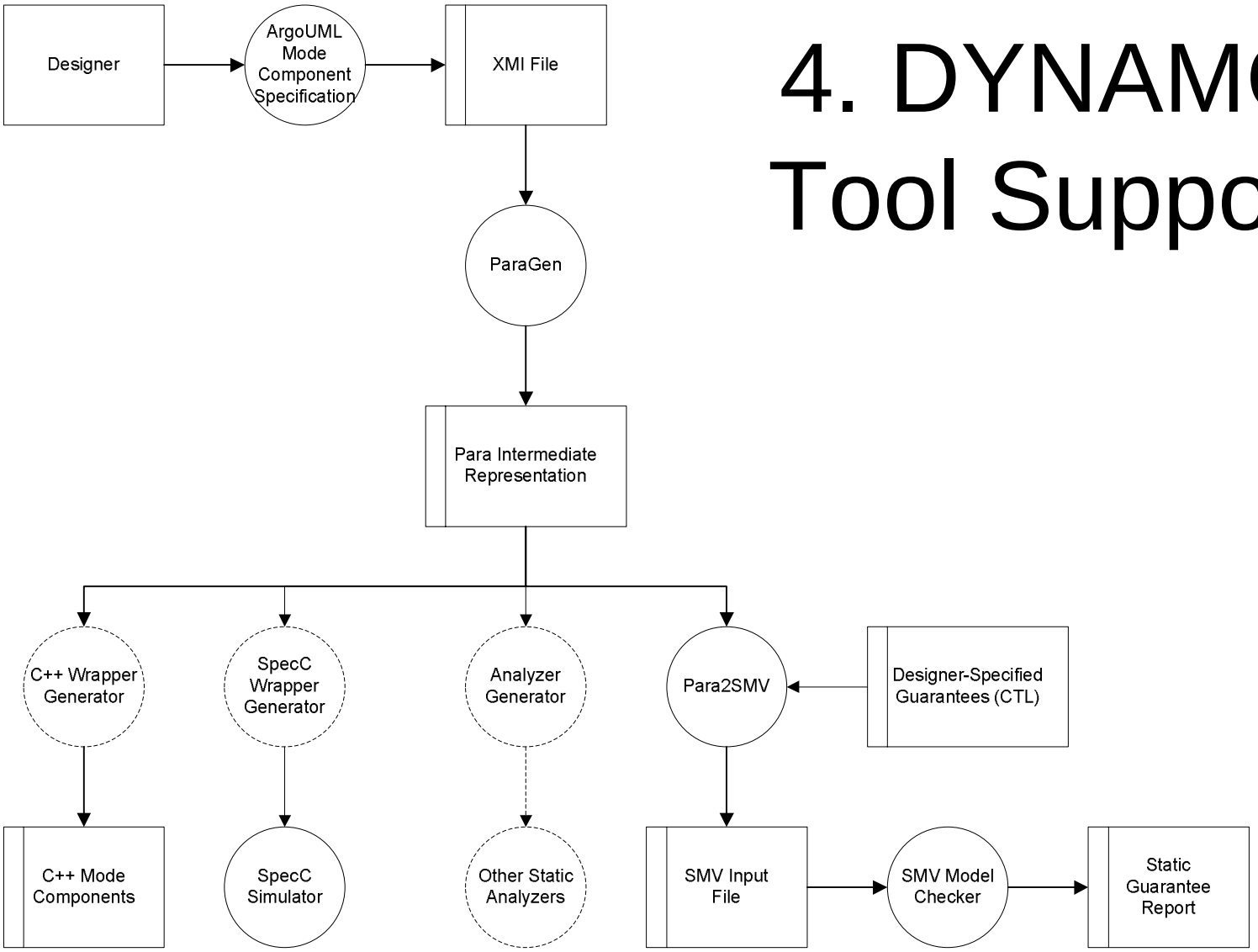
Separate Component / Mediator



- First-class invariant
- All components are black boxes
- Knowledge of component implementations required

⏭

4. DYNAMO Tool Support



Tools

- **COGITO** - OCL -> C++ wrapper generator
- **PLAT** - Product Line Assembly Tool (GUI)
- **Para:** Intermediate representation and API that provides easy access to UML designs
- **PARAgen** - ArgoUML (XMI) to PARA converter (C and XSLT versions)
- **PARA2SMV** - converted designs expressed in Para into the input format of SMV, a symbolic model checker.
- **SMVModel** - C++ library for generating SMV
- **PARAVisitor** - C++ library for walking PARA models

Future Work

- Finite differencing optimizations (optimized access to container classes)
  - *Data transformers*
- Comparison with use of metaobject protocol
  - OpenC++; alternative invariant maintenance mechanisms
- Run-time support for invariant maintenance with concurrent processes

Related Work

- Shaw and Garlan: Implicit invocation design space
- Aksit *et al.:* Composition filters
- Smaragdakis and Batory: Mixin layers
- VanHilst and Notkin: Role-based collaborations
- Sullivan and Notkin: Mediators
- Taylor *et al.:* Listening Agents
- DeLine: Flexible packaging
- Medvidovic *et al.:* Modeling architectures in UML
- Abowd and Dix: Integrating status and events

Project Links

- Project web page
  - http://www.cc.gatech.edu/dynamo
- Contact
  - spencer@cc.gatech.edu