The Command Pattern and the Strategy Pattern

Based on Gamma et al.

Command Pattern

- **Idea:** Represent actions (commands) with objects.
- "Command" objects are registered with "Invoker" objects
- Command objects know *what* to do
- Invoker objects know *when* to do commands
- Neither class depends on the other
- **Main consequences:**
    - The coding of an action is decoupled from its sequencing.
    - The Invoker class is reusable (often part of a framework).

Example (Commands in java.awt)

- Commands implement interface `java.awt.event.ActionListener`
- A command (inner) class

```
class LoadAction
    implements java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            loadFile(e);
        } }
```

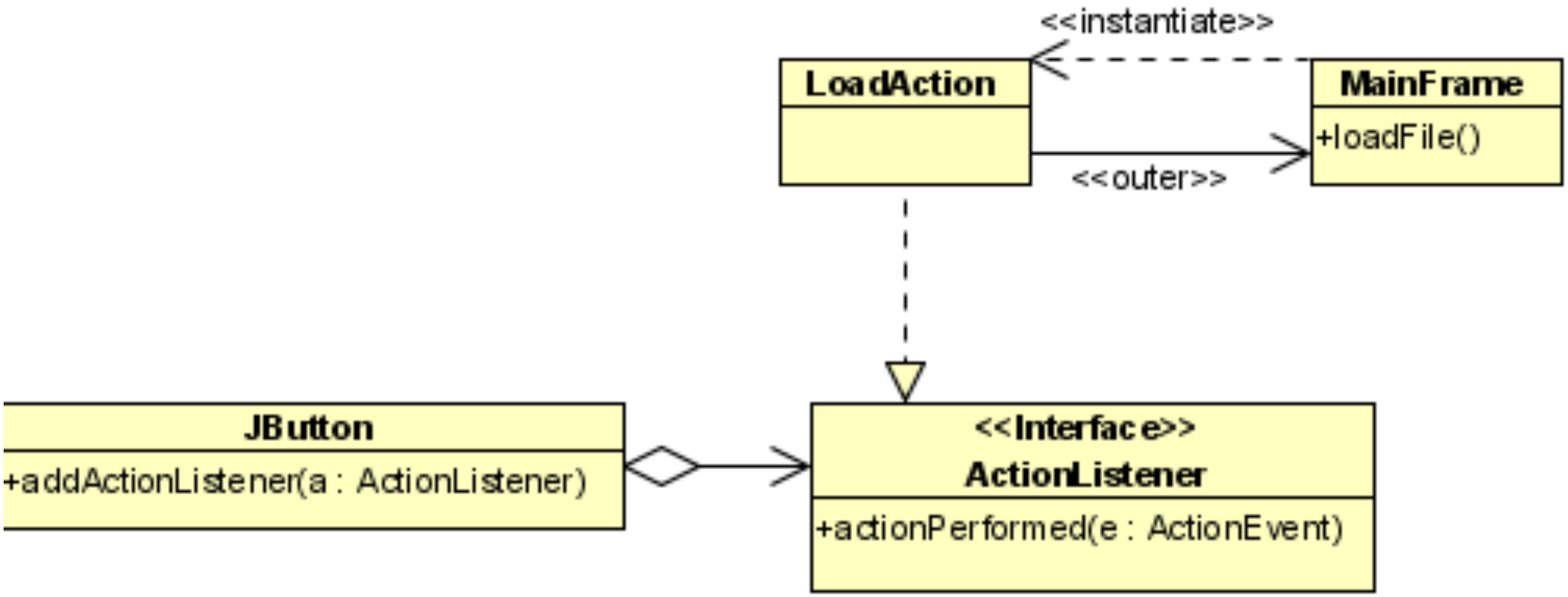
Example (Commands in java.awt)

- Command objects are passed to invokers such as buttons and menu items

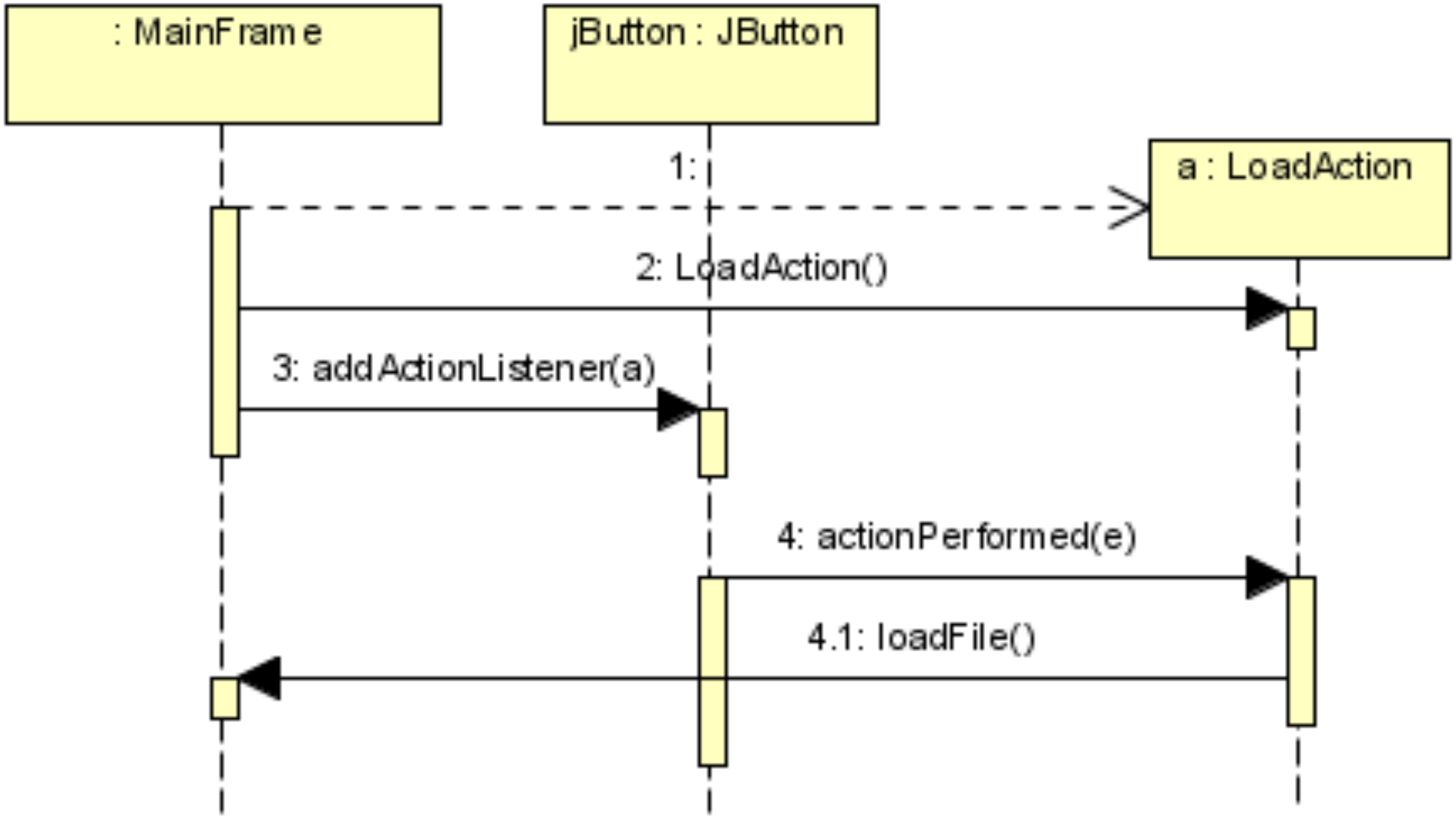
```
ActionListener loadAction = new LoadAction() ;
loadMenuItem.addActionListener( loadAction ) ;
loadToolBarButton.addActionListener( loadAction ) ;
```

- Invokers call their action listeners.

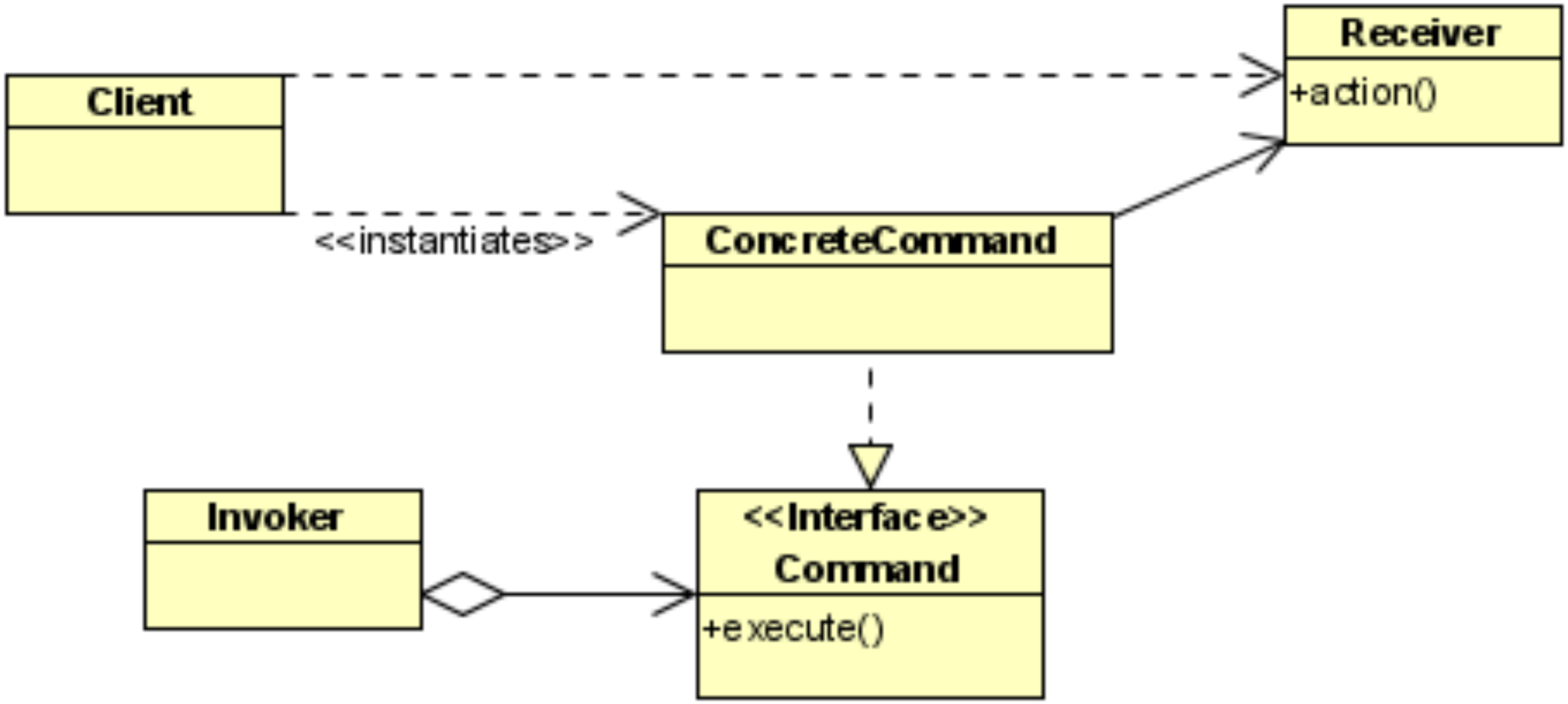
Example (Commands in java.awt)



Example (Commands in java.awt)

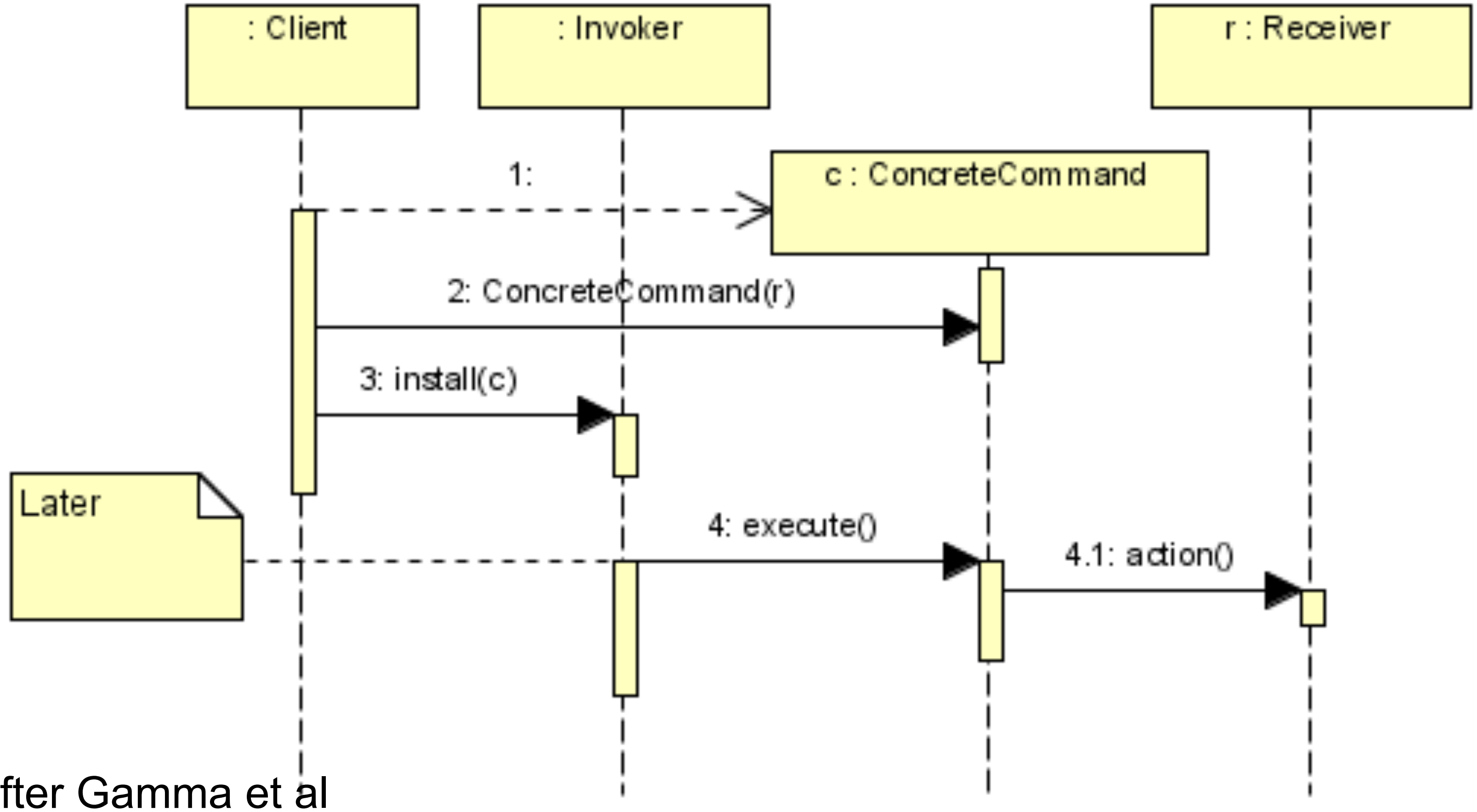


Pattern Structure



After Gamma et al

Pattern Collaboration



After Gamma et al

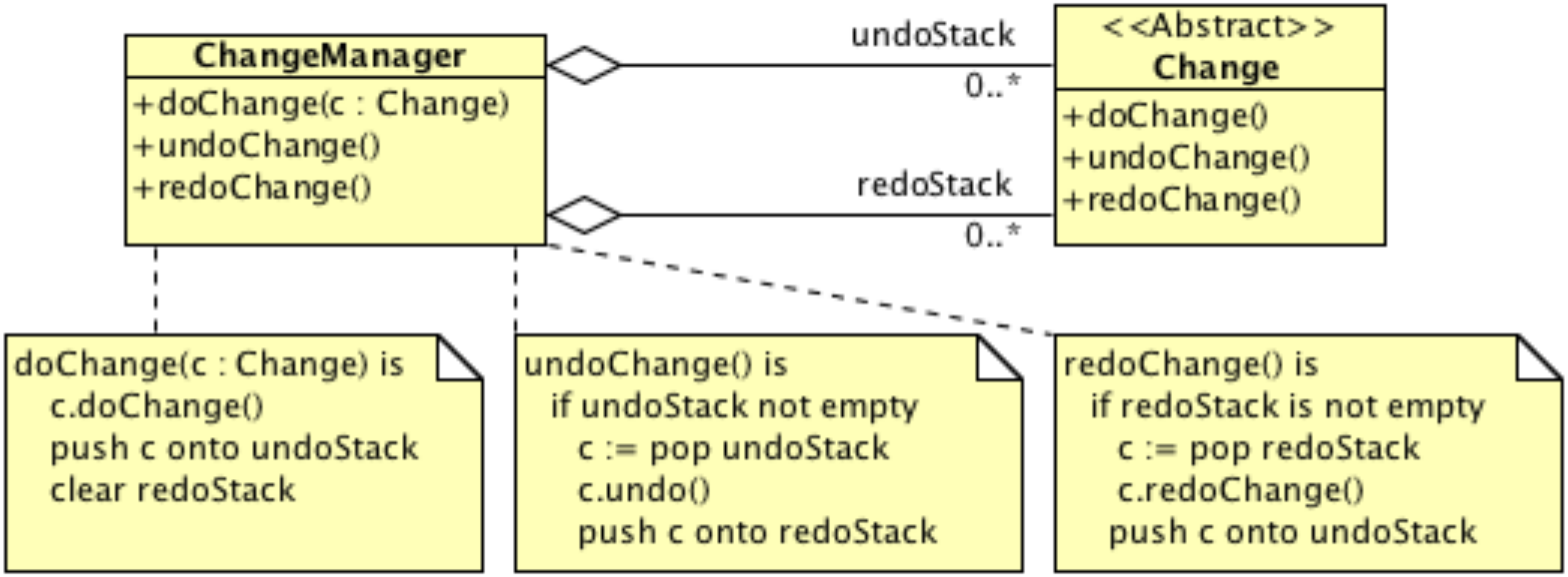
Consequences

- Invocation is decoupled from action
- (When is decoupled from what)
- Actions are data. They can be stored, moved, extended.
- Multiple invoker classes can be mixed and matched with multiple command classes.
- Commands can be aggregated to form composite commands. (E.g. to form macros.) See the Composite and Interpreter patterns.

Undoable Commands

- Objects associated with buttons etc create Change objects and send them to a ChangeManager.
- Each Change object supports doChange, undoChange, and redoChange.
- The ChangeManager sends “doChange” to the Change and adds it to an undoStack.
- The ChangeManager supports undoChange and redoChange.

Undoable Commands (cont.)



```
doChange(c : Change) is
    c.doChange()
    push c onto undoStack
    clear redoStack
```

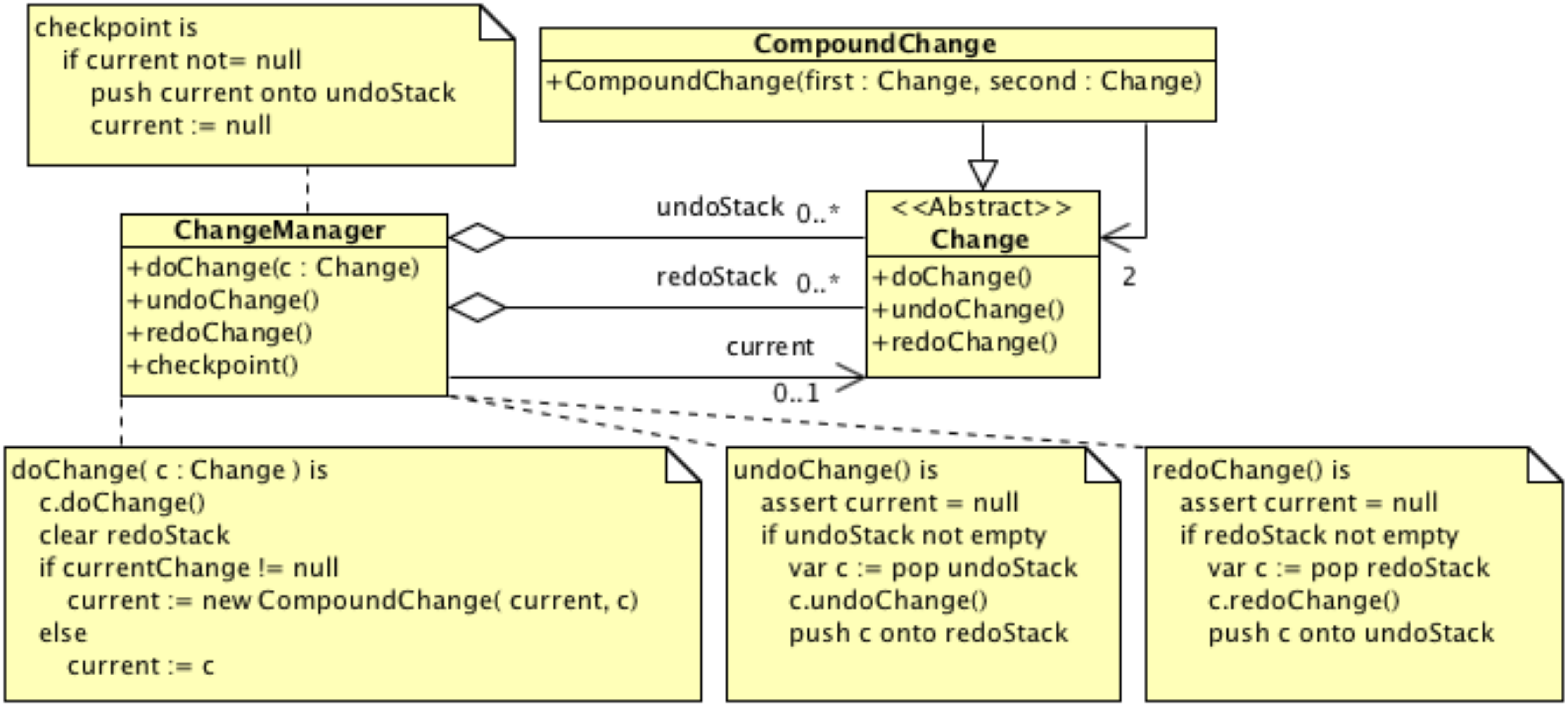
```
undoChange() is
  if undoStack not empty
    c := pop undoStack
    c.undo()
    push c onto redoStack
```

```
redoChange() is
  if redoStack is not empty
    c := pop redoStack
    c.redoChange()
   push c onto undoStack
```

Compound changes

- A problem with this scheme is that it forces the Changes to match UI cycles (one change per user interaction).
- Suppose we allow multiple Changes to be applied per UI cycle.
- At the end of each cycle the UI calls “checkpoint”
- Between checkpoints, compound changes are built.

Compound changes (cont.)



```
checkpoint is
    if current not= null
        push current onto undoStack
        current := null
```

CompoundChange

+CompoundChange(first : Change, second : Change)

ChangeManager

+doChange(c : Change)
+undoChange()
+redoChange()
+checkpoint()

<<Abstract>>
Change

+doChange()
+undoChange()
+redoChange()

```
doChange( c : Change ) is
    c.doChange()
    clear redoStack
    if currentChange != null
        current := new CompoundChange( current, c)
    else
        current := c
```

```
undoChange() is
    assert current = null
    if undoStack not empty
        var c := pop undoStack
        c.undoChange()
        push c onto redoStack
```

```
redoChange() is
    assert current = null
    if redoStack not empty
        var c := pop redoStack
        c.redoChange()
        push c onto undoStack
```

Example: Compilation in Turtle-Talk

- The turtle-talk compiler is intended to be reusable with different sets of “built-in” entities: subroutines, types, and constants.
  - For example, in the “Maze game”
    - built-in types include “bool” and “direction”.
    - built-in constants include “true”, “false”, “up”, “right”, “down”, and “left”.
    - built-in subroutines include
      - wall(d : direction) : bool
      - go(d : direction)
      - and many others

Compilation Example (cont.)

- The compiler does not depend on knowledge of these built-in entities. It is thus reusable.
- Each entity is represented by an entry in a table (the symbol table) that maps its name to a `SymbolTableEntry`.
- For constants, functions, and procedures, each `SymbolTableEntry` has a `CodeGenerationRule` object
- The `CodeGenerationRule` objects are command objects.

Compilation Example (cont.)

- Each `CodeGenerationRule` has a method

```
public void apply( int numberOfArgs,
                   Analyser analyser,
                   CodeEmitter codeEmitter )
       throws TurtleTalkException ;
```

  responsible for:

  - checking correct usage (right number and types of parameters) – via analyser
  - indicating return type – via analyser
  - generating code -- via codeEmitter

Compilation Example (cont.)

Example: the “up” constant of type “direction”.

```
CodeGenerationRule upCGR = new CodeGenerationRule() {
        public void apply( int numArgs,
                           Analyser analyser,
                           CodeEmitter codeEmitter )
        throws TurtleTalkException
        {
          analyser.check(numArgs==0, "args after constant" );
          codeEmitter.emitPush( Maze.UP );
          analyser.push( DIR_TYPE ); } } ;
```

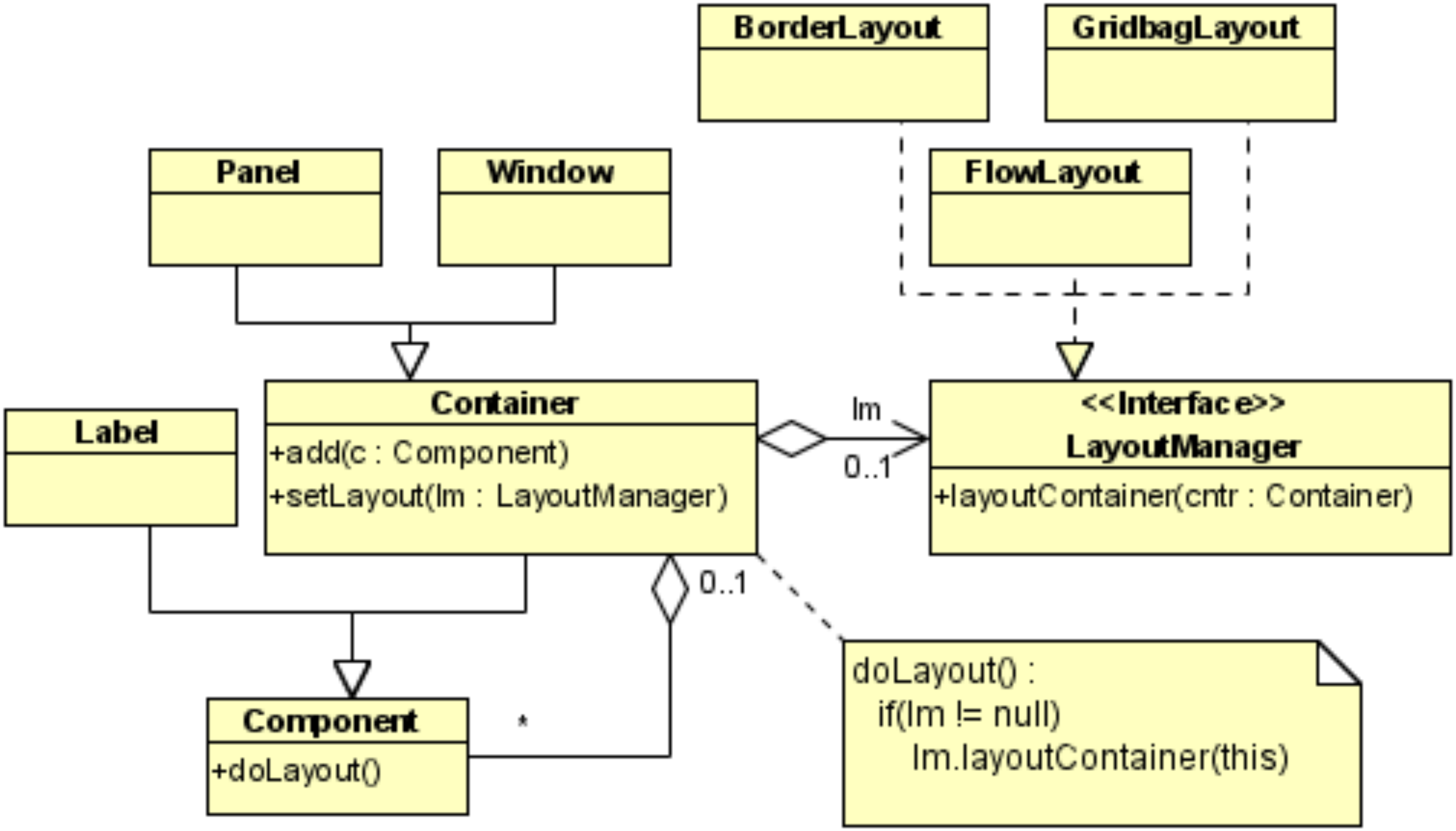
Compilation Example (final)

- When the compiler (invoker) encounters a function call or a procedure call, it
  - looks up the subroutine in the symbol table
  - emits code for the arguments
  - calls the `apply` method of the associated `CodeGenerationRule`.
- References to constants are similar.
- (The Teaching Machine also uses CGRs extensively)

Strategy Pattern

- **Idea:** Represent strategies (policies) with objects.
- Specialize general purpose classes by supplying them with strategy objects.

Example from the AWT

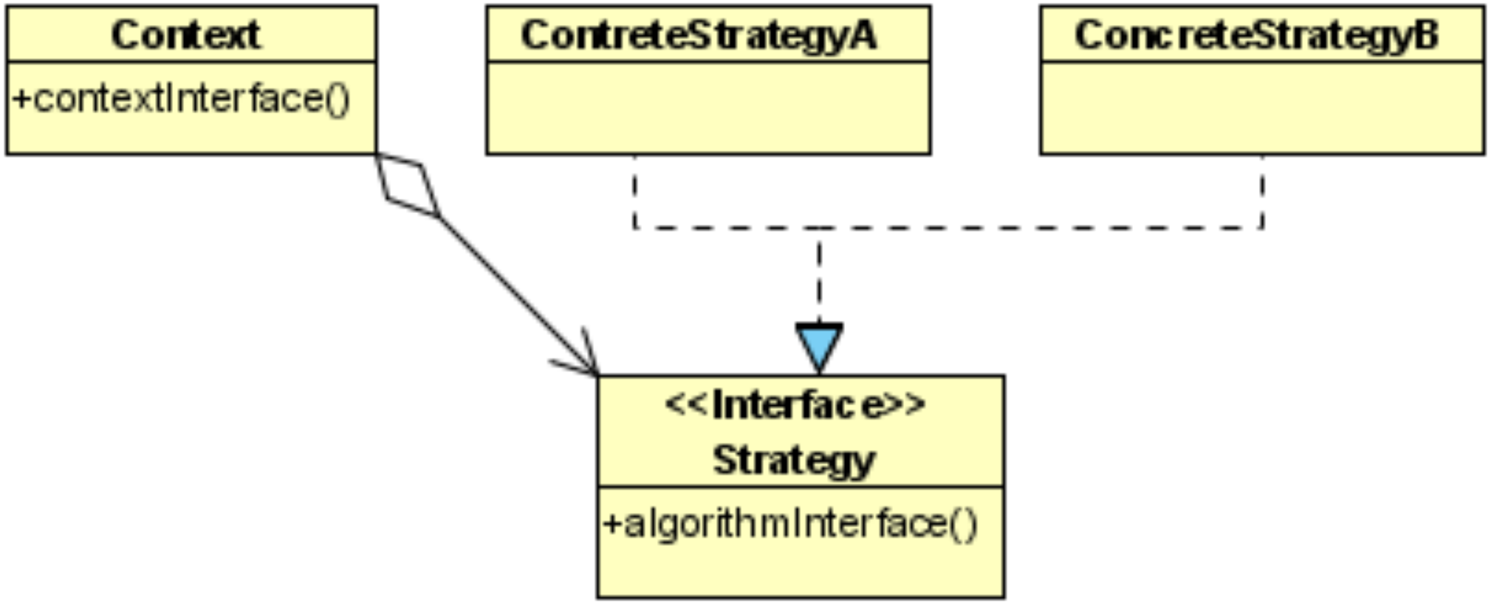


doLayout() :
  if(lm != null)
    lm.layoutContainer(this)

Example from the AWT

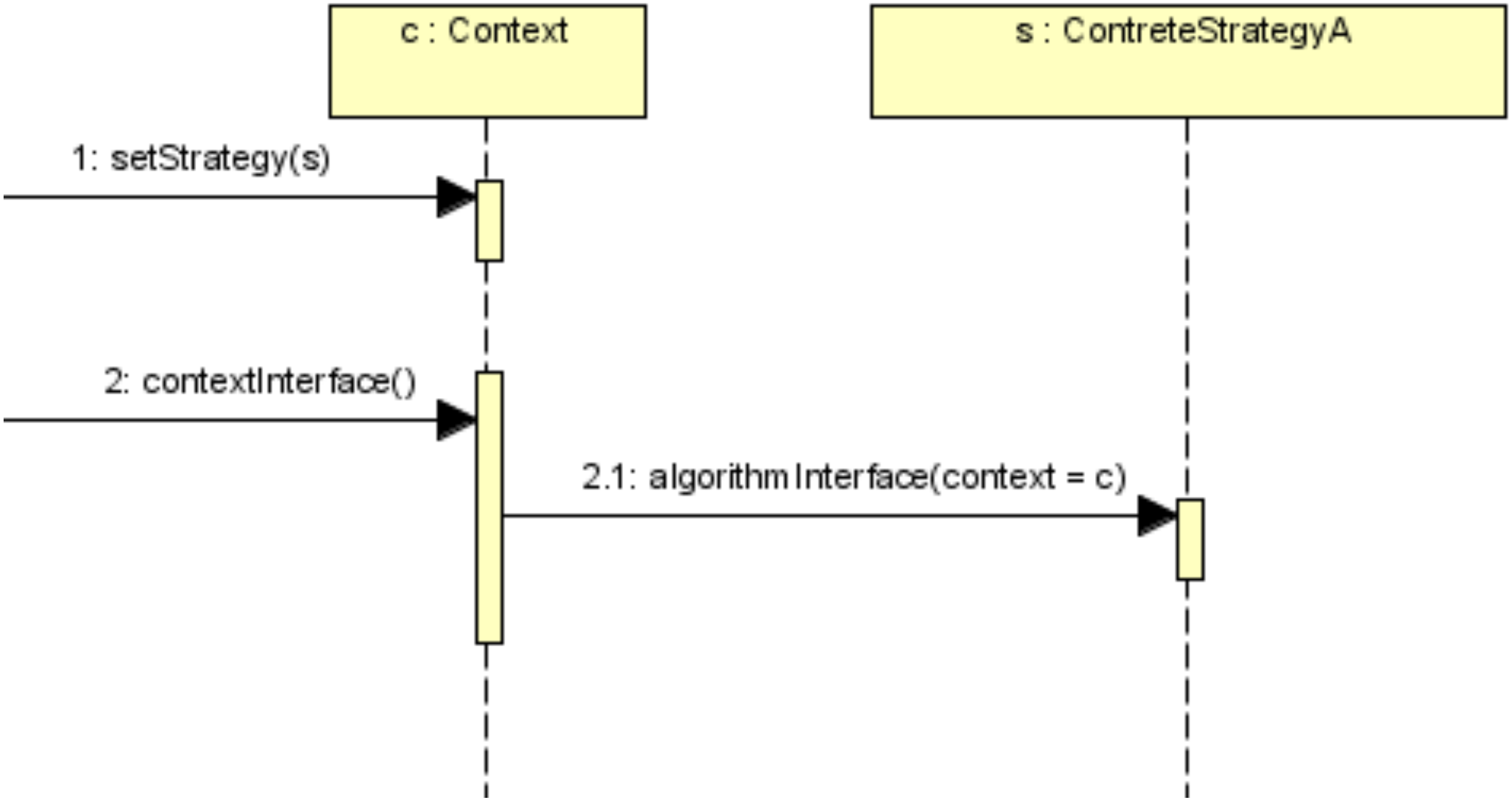
- Container classes may delegate to their layout manager to arrange their components.
- Clients can set the layout managers allowing mix-and-match combinations.
- Each new layout manager class can be used with any container class.
- Each new container class can be used with any layout manager class. (In theory at least.)

Strategy Structure



After Gamma et al

Strategy Collaboration



Consequences of the Strategy pattern

- **Main consequences:**
  - Aspects of a class's behaviour can be modified by the choice of a strategy object.
  - The client can choose how to combine strategy with context.
  - Objects can appear to change class at runtime
  - Strategies may be stored and looked up.
  - Alternative to conditional statements.
  - Orthogonal class hierarchies.
    - Strategies can form a class hierarchy orthogonal to the hierarchy of clients
  - Alternative to (multiple) inheritance.

Aside: Use inheritance rather than conditionals

- A hypothetical design

```
class Container { …
    public void doLayout() {
        switch( this.layoutKind ) {
        case BorderLayoutKind : … break ;
        case FlowLayoutKind : … break ;
        case GridbagLayoutKind: … break ; } … }
```

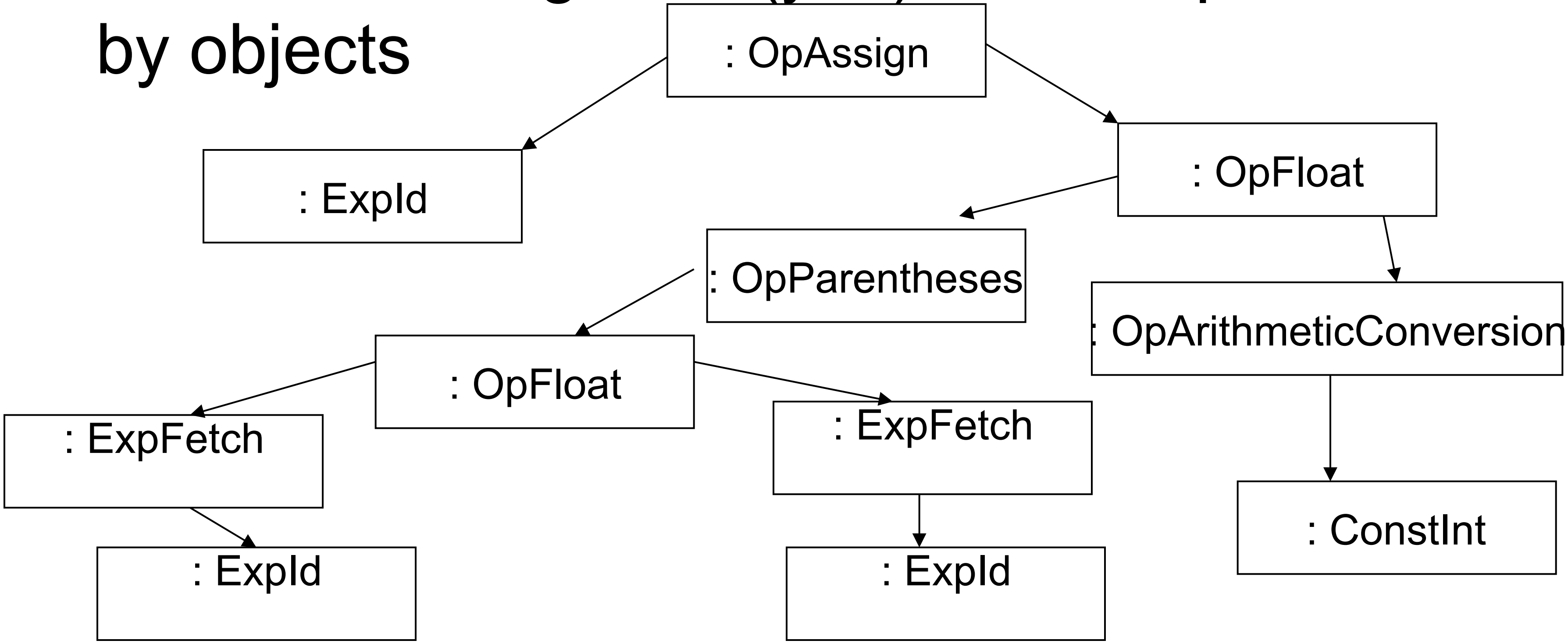
- Clearly this is not extensible.
- Any time you use conditional commands, ask your self if there is an OO alternative.

Aside: Delegation vs inheritance

- Delegation is often preferable to inheritance, as the delegate can be chosen by the instantiator and even vary across time.
- In a single inheritance language, delegation provides an alternative to multiple inheritance
- Consider a design with inheritance hierarchies of *n* concrete contexts and *m* concrete strategies. There are *m*n* combinations possible for the price designing *m+n* concrete classes.

Example from the Teaching Machine

- Expressions are represented by nodes that form a tree. E.g. “x = (y+z) / 2” is represented by objects



: OpAssign

: ExpId

: OpFloat

: OpParentheses

: OpArithmeticConversion

: OpFloat

: ExpFetch

: ExpFetch

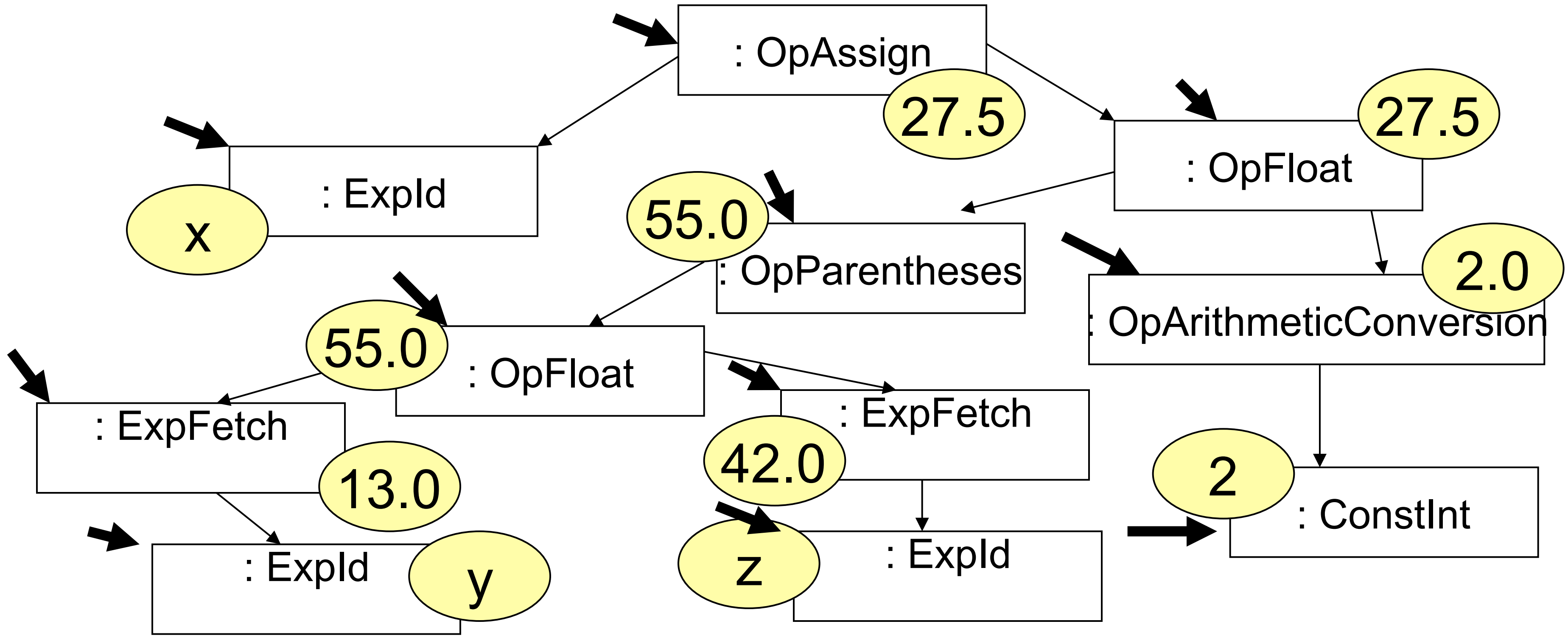
: ConstInt

: ExpId

: ExpId

Example from the Teaching Machine

- Expressions are evaluated by alternately:
  - “Selecting” a ready node
  - “Stepping” the selected node


: OpAssign
27.5
: ExpId
x
: OpFloat
27.5
55.0
: OpParentheses
: OpArithmeticConversion
2.0
55.0
: OpFloat
: ExpFetch
13.0
: ExpFetch
42.0
: ExpId
y
z
: ExpId
2
: ConstInt

Example from the Teaching Machine

- Expression nodes vary along multiple axes
  - Number of children
  - Order of evaluation of children & self (selection)
  - Execution algorithm (stepping)
  - Conversion of self to string for display
- The first version of the TM tried to use inheritance to accommodate these multiple axes of variation.
- The result was a deep and complex inheritance hierarchy that still did not eliminate duplication

Example from the Teaching Machine

- The TM was redesigned so
  - Each subclass of node knows two strategy objects.
  - One strategy determines the order of evaluation of children & self. (Selection)
  - One strategy determines the execution algorithm (Stepping)
  - Both are set in the constructor

Example from the Teaching Machine

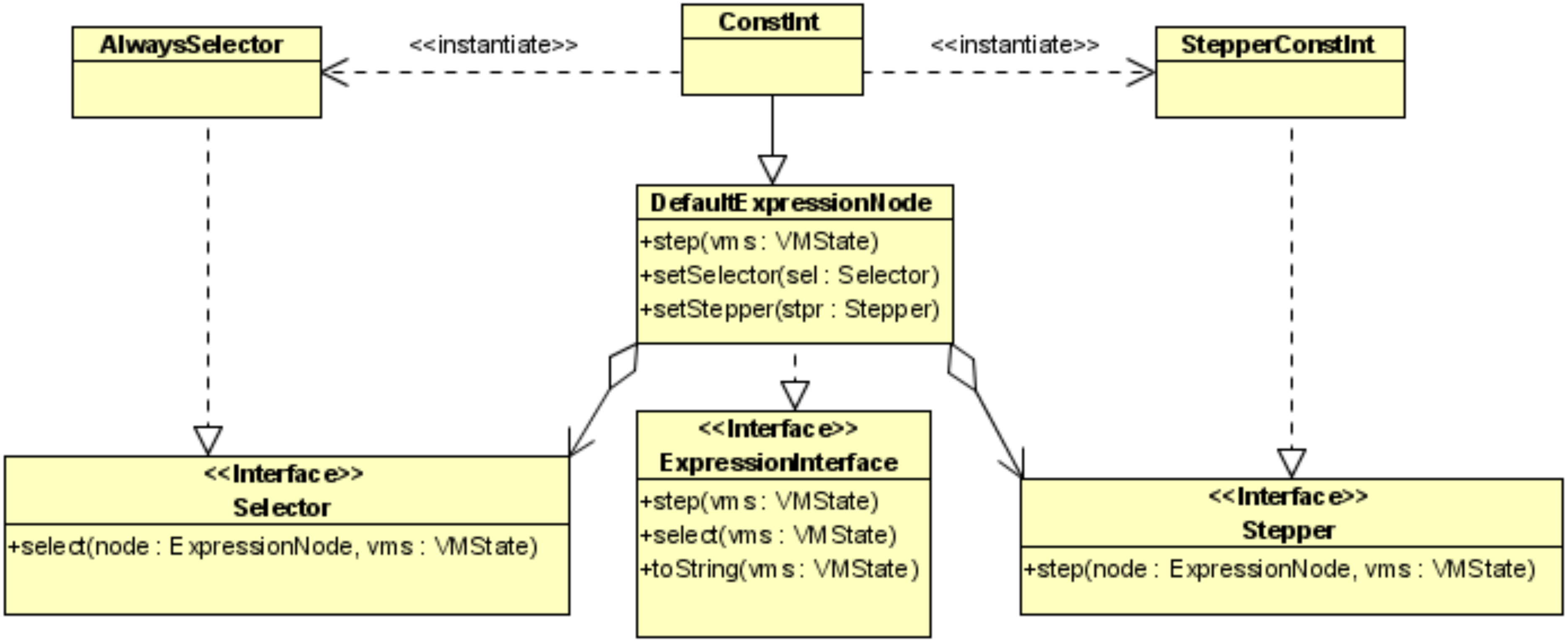
- Consider execution (stepping).
- The step method for nodes delegates to a Stepper object:

```
public void step( VMState vms ) {
        Assert.check(stepper != null) ;
        stepper.step(this, vms) ;
    }
```

- The stepper for ConstInt:

```
public void step( ExpressionNode nd, VMState vms ) {
    create an object representing the integer
    associate the node, nd, with this new object }
```

Example from the Teaching Machine



<<instantiate>>

<<instantiate>>

AlwaysSelector

ConstInt

StepperConstInt

DefaultExpressionNode
+step(vms : VMState)
+setSelector(sel : Selector)
+setStepper(stpr : Stepper)

<<Interface>>
Selector
+select(node : ExpressionNode, vms : VMState)

<<Interface>>
ExpressionInterface
+step(vms : VMState)
+select(vms : VMState)
+toString(vms : VMState)

<<Interface>>
Stepper
+step(node : ExpressionNode, vms : VMState)

Example from the Teaching Machine --- Caveat

- The setting of strategies is done in the contexts' constructors, not by the client
- Thus this use of the strategy pattern in the TM is strictly internal to the node package. I.e. the strategy pattern is used only as an implementation technique.
- By contrast the strategy pattern usually provides the client with a selection of contexts and strategies and the ability to extend either.
- The TM approach means the client is provided with many context classes, but no strategy classes.

Retrospect on Strategy in the TM

- In retrospect, the use of strategies for selection was highly successful. A small number of strategies are reused in various contexts
- The use of Stepper strategies was less successful. Stepper subclasses and ExpressionNode subclasses were in almost a one-one and onto correspondence, so the benefit was negligible.
- However as the extra complication was internal to the node package, the cost was contained. I.e. no cost was paid by client code.
- Furthermore we did make use of stored Steppers to implement built-in function calls --- an unexpected benefit.