Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling Contex-Aware Behavior by Ambient ASMs

Behavioral Programming Patterns Case Study

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 4.3 of Modeling Companion
http://modelingbook.informatik.uni-ulm.de

Theme: Modeling context-aware system behavior

General question: How to gently model contexts and programs whose run behavior depends on the context in which the program is executed?

- NB. Often contexts are called environments, not to be confused with env understood as interpretation of free variables

In this lecture we use behavioral programming patterns

- source: E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns (Addison-Wesley 1994)

as examples to model context-aware behavior by ambient ASMs

- we recapitulate motivation and definition of ambient ASMs

General idea: *use parameterization* to model context dependency.

The role of parameterization for ASMs

Parameterizing ASM functions and rules permits to directly model:

- partitioning/isolating of states and distributed computations
  - of agents which concurrently execute in heterogeneous contexts
- various forms of information hiding (encapsulation of memory)
  - statically: scope, module, package, library, etc.
  - dynamically: executing agents, threads, process instances, etc.
- patterns of programming, of communication and of control flow
- mobility (environments where agents can move)
- modularity (of specs and property verifications)

Exploit *simplicity of semantical foundation* of parameterization:

$$f(x) = f_{params}(x)$$

in particular when used with implicit (hidden) parameters, supporting conventional implicit oo parameterization $this.f(x) = f(x)$

Ambient ASMs to explicitly support env-sensitive actions

Idea: enrich ASMs by an abstract ambient parameter

- with respect to which the terms involved in a step are evaluated
- for ambients (environments) at whatever level of abstraction
- which can be created, modified, deleted, also at run time

Syntactical construct: **amb** exp **in** P

where to achieve generality in the widest terms

- exp is any expression (term)
- P is any (already defined) ASM program

Intended behavior (see definition below):

- PUSH the $eval(exp, S, env, amb)$ of the newly declared ambient expression to the current ambient
- execute P in the new ambient

Function classification is extended by ambient (in)dependent fcts.

Ambient dependent functions and their evaluation

$AmbDependent(f)$ (wrt Amb) **iff**

forsome $a, a' \in Amb$ **with** $a \neq a'$ **forsome** x $f(a,x) \neq f(a',x)$

Otherwise f is called $AmbIndependent$. We also say *environment (in)dependent*, hoping that Ambients will not be confused with Environments (interpretation of free variables in a state).

- Case $AmbDependent(f)$:

  $eval(f(t_1, \ldots, t_n), S, env, amb) =$
  $f_S(amb, eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$

  Here f_S is turned into a family of possibly different functions $f_{S,amb}$.
- Case $AmbIndependent(f)$ (unchanged interpretation of f):

  $eval(f(t_1, \ldots, t_n), S, env, amb) =$
  $f_S(eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$

Semantics of ambient ASMs

To avoid a signature blow up by dynamic ambient nesting, we adopt Simone Zenzaro's idea (PhD Thesis, Pisa 2016) to

treat amb as a stack

where new ambient expressions are pushed (passed by value)

- so that for each $f^{(n)}$ one extension $f^{(n+1)}$ suffices which offers an additional argument position for the interpretation of $f^{(n)}$ in a given ambient.

The body of **amb** exp **in** P is then executed with the new stack value.

$Yields(\textbf{amb}\ exp\ \textbf{in}\ P, S, env, amb, U)$ **if**

$$Yields(P, S, env, \text{PUSH}(eval(exp, S, env, amb), amb), U)$$

- NB. Often the execution of P (read: the interpretation $f_{S,amb}$) depends only on the top of the stack, i.e. on $eval(exp, S, env, amb)$ (*flat ambient ASMs*, see JCSS 2012).

Special case: flat ambient ASMs

Only the last declared (most recent) ambient is kept

- instead of an ambient stack (of nested ambients) to which each newly declared environment is PUSHed
  - This was the original definition of ambient ASMs in JCSS 2012, which is generalized by the stack interpretation of ambients.

For flat ambients it suffices to bind the value of the expression in the ambient declaration

- which is computed in the current state (in the current ambient)

to a logical variable, say $curamb$:

$$(\textbf{amb}_{\textbf{flat}}\ exp\ \textbf{in}\ P)^* = (\textbf{let}\ curamb = exp^*\ \textbf{in}\ P^*)$$

- where exp^* is obtained from exp by replacing the names f of ambient dependent functions in exp by f_{curamb}

Inductive transformation of flat ambient ASMs

- *transformation of terms*:
  - for $AmbIndependent(f)$ define:

    $(f(t_1, \ldots, t_n))^* = f(t_1^*, \ldots, t_n^*)$

    - logical variables and names of parameterized rules are classified as ambient independent
  - for $AmbDependent(f)$ define:

    $(f(t_1, \ldots, t_n))^* = f(curamb, t_1^*, \ldots, t_n^*)$
- *transformation of ambient ASM rules* (to standard ASM programs):
  - for update rules define:

    $(f(t_1, \ldots, t_n) = t)^* = (f(t_1, \ldots, t_n)^* := t^*)$
  - for ambient rules define (eliminating $\mathbf{amb_{flat}}$):

    $(\mathbf{amb_{flat}}\ exp\ \mathbf{in}\ P)^* = \ (\mathbf{let}\ curamb = exp^*\ \mathbf{in}\ P^*)$
  - for the other rules use induction on ASM programs, e.g.

    $(\mathbf{let}\ x = t\ \mathbf{in}\ P)^* = \ (\mathbf{let}\ x = t^*\ \mathbf{in}\ P^*)$

Traditional classification of oo design patterns

- by purpose:
  - structural patterns concerning composition of classes and objects
  - creational patterns concerning creation of objects upon class instantiation
  - behavioral patterns concerning interaction of classes and objects (flow of control, actions, communication and cooperation)
- by scope:
  - class patterns dealing with static (compile-time relevant) composition of (sub)classes (inheritance, accessibility, use relations)
  - object patterns dealing with dynamic composition of run-time objects

Using the ASM Method we lift the object-oriented subclassing and inheritance view to the more general (mathematically precise) ASM refinement concept, using ambient ASMs.

Behavioral classification of oo design patterns

We illustrate the parameterization of (sub)machines or operations by implicit arguments by the Delegation pattern and some of its refinements:

- Delegation with various refinement examples
  - Structural examples: Template, Proxy
    - Proxy with remote, virtual and protection version
  - Behavioral examples: (Chain of) Responsability, Bridge
  - Incremental refinement: Decorator

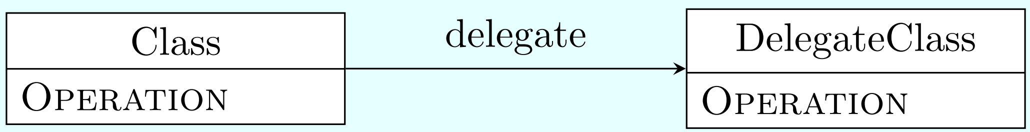
Ambient ASM definition of Delegation Pattern behavior

Delegation 'separates' instantiations of an Operation

- via an abstract *Class* interface Operation from implementations of Operation in a concrete *DelegateClass*[1]

s.t. at run-time for a call of Operation(x) one can determine a *delegate* in *DelegateClass* to execute its Operation instance

- i.e. the implementation provided in *DelegateClass*



Delegate(Operation, *delegate*)(x) =

  amb *delegate* **in** Operation(x)

NB. The pattern permits multiple *DelegateClass*es.

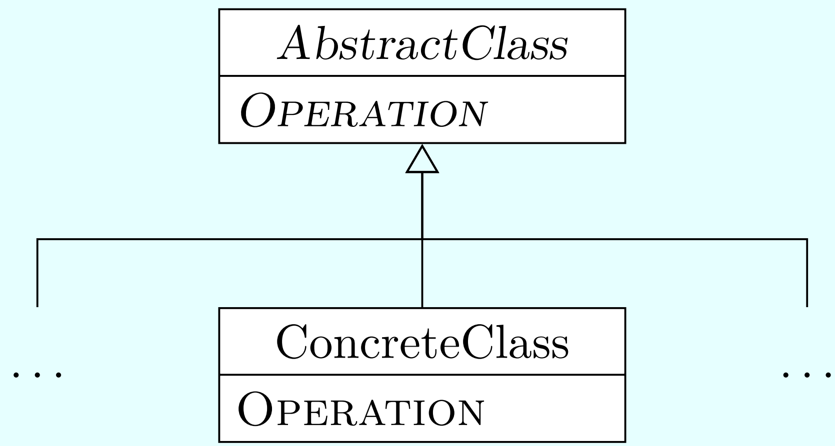
Pattern variations result from different ways to define *delegate*.

Instances of DELEGATE pattern

Alternatives to define *delegate*:

- *internally*: define *delegate* as a location
  - in the abstract *Class* (*Bridge* pattern)
  - in some dedicated subclass to 'provide a placeholder for another object' so that delegate is 'the real object that the proxy represents' (*Proxy* pattern and its refinements)
- *externally*: define *delegate*
  - statically:
    - determined by the class structure (*Template* pattern)
    - determined by a data-structure related function (like the chain traversal function in *ChainOfResponsibility* pattern)
  - dynamically (*Responsibility* pattern (using run-time determined *selection* function), *Bridge* and emph Decorator patterns)

Template pattern requirements



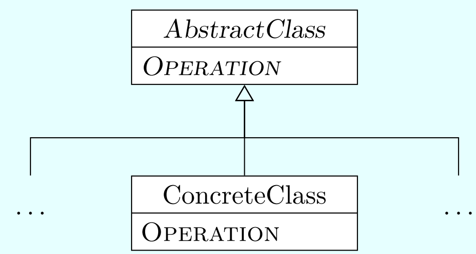
Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Idea: use $delegate$ as denoting a subclass $ConcreteClass$ of $AbstractClass$, determined by the static subclass structure.

TEMPLATE **pattern: an instance of** DELEGATE

NB. In an ASM, arbitrary ambient expressions are permitted, e.g. $ConcreteClass$.

TEMPLATE = DELEGATE **where** $delegate = ConcreteClass$



OPERATION: 'the skeleton of an algorithm' (an 'Application'), may call some abstract *PrimitiveOperations*

- interfaces in *AbstractClass* are implemented (as individual 'MyApplication') in a $ConcreteClass$
  - which provides its interpretation $op(ConcreteClass, x)$ of the abstract *PrimitiveOperations* $op(x)$ 'to carry out subclass-specific steps of the algorithm': an ASM refinement of type (1,1)

RESPONSIBILITY **pattern requirements**

Goal:

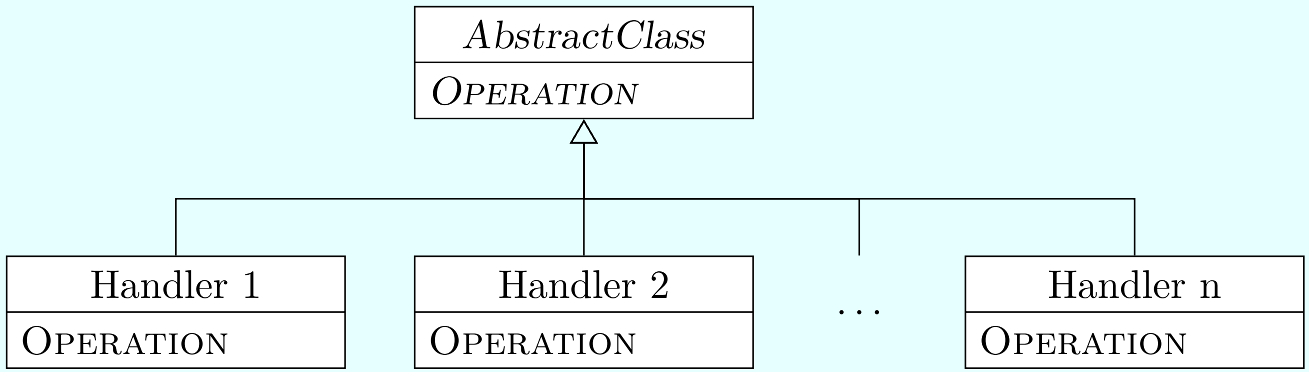
> avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request

in particular when static association of caller with delegate is impossible.

It suffices to $select$ a delegate among $Receiver$s in subclasses $Handler\ 1, \ldots, Handler\ n$ of the $AbstractClass$ which $CanHandle$ the input $request$

RESPONSIBILITY pattern: a data-refinement of DELEGATE

Responsibility Pattern Class Structure:



Idea: *select* a delegate among *Receiver*s in subclasses *Handler i* ($1 \leq i \leq n$) of the *AbstractClass* which *CanHandle* the input *request*

$$\text{RESPONSIBILITY} = \text{DELEGATE}$$

$$\textbf{where } delegate = select(PossibleHandler(x))$$

$$PossibleHandler(request) = Receiver(request) \cap Handler(request)$$

$$Handler(request) = \{o \mid CanHandle(o, \text{OPERATION})(request)\}$$

$\textsc{ChainOfResponsibility}$: **data-refined** $\textsc{Delegate}$

Requirements:

‘chain the receiving objects and pass the request along the chain until an object handles it ... the handler should be ascertained automatically’

This can be achieved by a data refinement:

- of the non-deterministic choice fct $select$ in $\textsc{Responsiblity}$
- to a deterministic function which with respect to an order $<$ (‘chain’) of $Receiver(request)$ provides the $first$ element that $CanHandle$ the input $request$

$\textsc{ChainOfResponsibility} = \textsc{Delegate}$

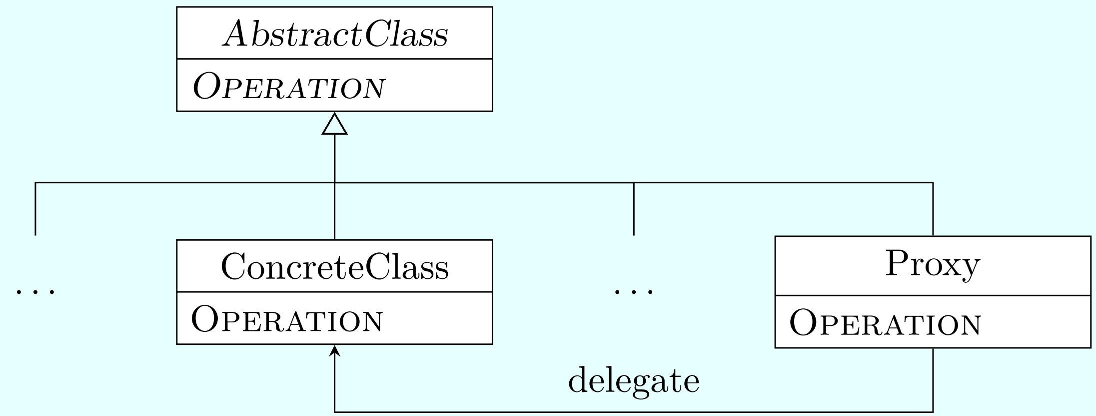
where $delegate = first(PossibleHandler(x))$

NB. To ‘ascertain’ the handler ‘automatically’ means to program the $delegate$ function. function

Proxy Pattern requirements and class structure

Proxy intended to 'provide a surrogate or placeholder for another object to control access to it' such 'that a Proxy can be used anywhere a RealSubject is expected'.

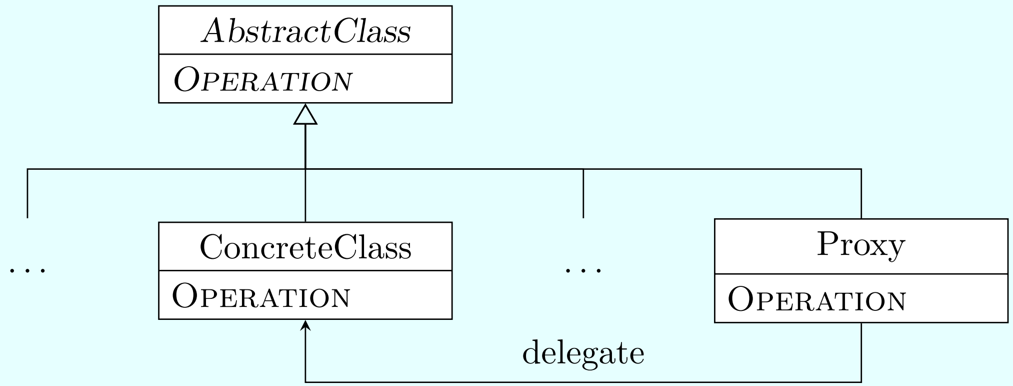
Class structure:



delegate becomes 'the real object that the proxy represents' and thus is renamed to *RealSubject*

- value of *delegate* is a *ConcreteClass* instance
  - kept in a placeholder location of the dedicated *Proxy* subclass

Proxy Pattern: a data refinement of DELEGATE



- Via *Proxy*, client calls of OPERATION are forwarded to *delegate*
  - which is passed as ambient parameter to the refinement of OPERATION
    - i.e. to its implementation in the *classOf*(*delegate*)

PROXY = DELEGATE **where** *delegate* denotes a location of *Proxy* with values in any subclass *ConcreteClass* of *AbstractClass*

Further data refinements of PROXY pattern

Refinements by constraints on where the values of *delegate* are stored or on the access to *delegate*:

- in a REMOTEPROXY the *delegate* location is required to be in a different address space
- in a VIRTUALPROXY the *delegate* value is cached via some operation CACHE(*delegate*, *request*) so that its access can be postponed
- in a PROTECTIONPROXY it is checked whether the caller has the permission to access the OPERATION in *classOf*(*delegate*)

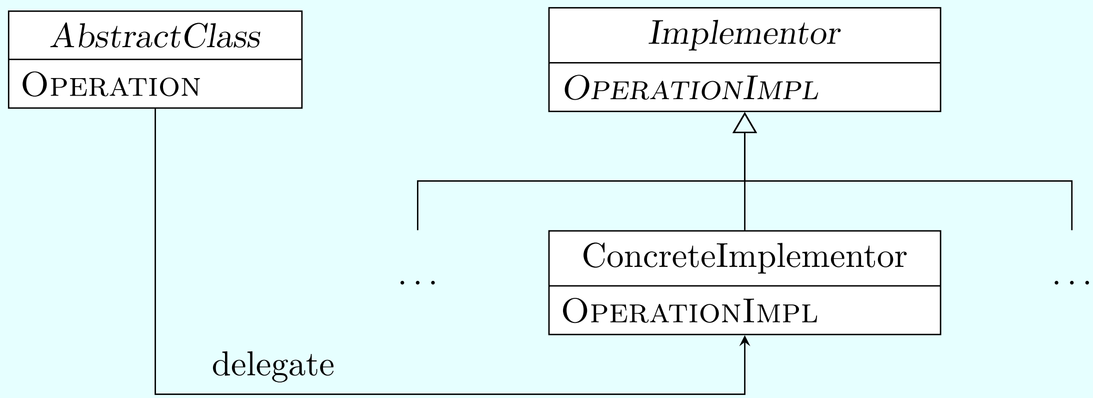
Example: LEADELECTection ASM built out of abstract components

PROPOSE, RECEIVEMSG, UPDATEKNOWLedge.

These components can be refined to compute

- the leader
- the leader plus notification of the termination
- the leader together with a shortest path to it

Bridge pattern: an instance of Delegate



delegate declared as *AbstractClass* location, its values are instances of outsourced implementing subclasses of an *Implementor* class

- with a link relating the interface Operation in *AbstractClass* to the interface OperationImpl in *Implementor* class

$$\textsc{Bridge}(\textsc{Operation}, \mathit{delegate}) = \textsc{Delegate}(\textsc{OperationImpl}, \mathit{delegate})$$

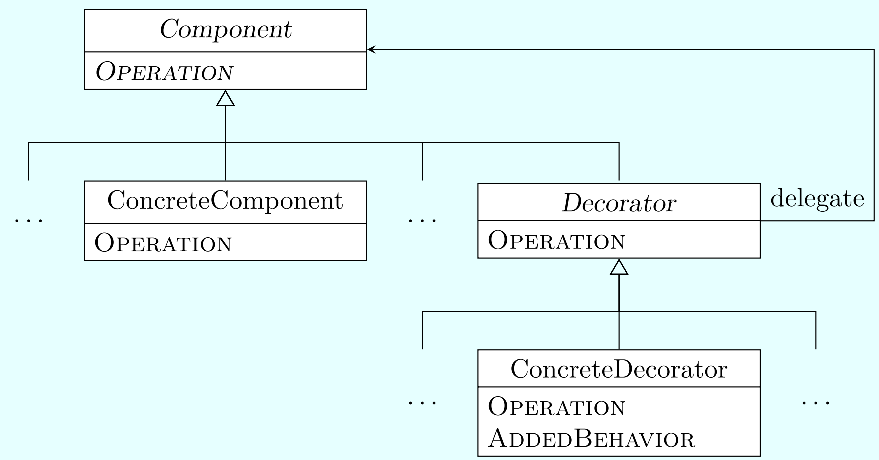
Run-time choices bw Operation refinements via updates of *delegate* replace static binding of implementations via class inheritance.

DECORATOR pattern requirements and class structure

Requirements:

'attach additional responsibilities to an object dynamically' as 'a flexible alternative to subclassing for extending functionality'

- leaving the OPERATION behavior of other class instances unchanged



Idea: keep 'a reference to a *Component* object' in a *delegate* location of a subclass *Decorator*, an 'interface for objects that can have responsibilites added to them dynamically'

- one *ConcreteDecorator* subclass for each ADDEDBEHAVIOR

DECORATOR pattern: instance of DELEGATE

As ASM program, DECORATOR is identical to DELEGATE:

DECORATOR(OPERATION, *delegate*) =
  amb *delegate* **in** OPERATION
    where *delegate* denotes a location of *Decorator*
      with values in any subclass *ConcreteDecorator* of *Decorator*

The difference is in how the parameterization of OPERATION by the environment is defined

- in DECORATOR, the implementation of OPERATION extends the *Component* OPERATION by ADDEDBEHAVIOR
- in DELEGATE, there is no extension of the implementation of OPERATION

NB. If the pattern only extends functionality without overriding the behavior of OPERATION, it represents an instance of incremental ('conservative') ASM refinements.

Work to be done

- model more complex not oo-programming-centric patterns, e.g.
  - J2EE Core Patterns (Alur et al., Prentice Hall 2003)
  - Patterns of Enterprise Application Architecture (Fowler et al., Addison-Wesley 2004)
  - Enterprise Integration Patterns (Hohpe & Woolf, Addison-Wesley 2004)
  - WS-CDL Service Interaction Patterns (Barros et al. 2005)
- define genuine modeling-patterns and their refinement

References

On ambient ASMs:

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de
  - Ch.4 contains other applications of ambient ASMs for contex-aware system models and further references
- E. Börger and A. Cisternino and V. Gervasi: Ambient Abstract State Machines with Applications.
  J. Computer and System Sciences 78.3 (2012) 939-959.

On oo programming patterns:

- E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns (Addison-Wesley 1994)

Copyright Notice