Dynamic Adaptation of Non-Functional Concerns

Pierre-Charles David & Thomas Ledoux

Ecole des Mines de Nantes

Outline

- Motivation
- General Approach
- Base Program and Join-Points
- Non-Functional Aspects
- Adaptive Weaving Specification
  - Dynamic Pointcut Definition
  - Context-Aware Weaving
- Conclusion

Motivation

- Software must be able to adapt to changes
  - in the execution environment
  - in the users' needs
- Most changes are not predictable at design and coding time
- Even when they are, taking them into account results in overly complex systems

General Approach

- We consider adaptations relative to the execution context's evolutions
- Distinguish *functional code* ...
  - pure business logic, does not depend on this context
- … from *non-functional services*
  - middleware-type services, depend on the context
  - ex: persistence and distribution QoS
- Services are applied to functional code

Example Scenario

- Domain: bank application
- Service: monitoring of «low» accounts
  - identify the corresponding `Account` instances
  - log all activities of these accounts
- Implementation adaptation:
  - if possible, store on local disk in XML (easy to analyze)
  - otherwise try on remote disk (but don’t use too much bandwidth)

Aspect-Oriented Programming

- An AOP system (AspectJ) consists in:
  - a base program
    - self-contained
    - contains *join-points*
  - aspect programs
    - define *pointcuts* as sets of join-points
    - attaches *advices* to these pointcuts

Applying AOP to our decomposition

- Functional code ⇔ Base program
- Non-Functional Services ⇔ Advices
- Adaptation Policies ⇔ Weaving Specification
- *Originality*: weaving specifications are:
  - externalized
  - context-aware
- The weaving can be adapted to changes in the execution context

Base Program & JoinPoints

- Programmers write standard Java
- Special compiler adds a wrapper to functional objects: `Container`
- Objects execution controled by `Container:`
  - intercepts method calls
  - exposes a *joinpoint*
  - accepts dynamic attachment of *advices*

Non-Functional Aspects

- Non-Functional code implemented by wrapper objects
  - act as generic *advices*
- Dynamically associated to `Container`s
- `Container` composes (sequentially) advices on methods interceptions

Dynamic Pointcut Definition

- Pointcuts:
  - by definition: a set of join-points
  - in our system: a group of functional objects
- Functional objects have dynamic *attributes*
  - some predefined: `className`
  - others reflecting their fields: `balance`
- Groups defined in intention
  - by a predicate over attributes
  - dynamically updated

Application Policies

- Domain Specific Language (DSL) to:
  - specify groups/pointcuts
  - bind adaptive aspect programs
- Syntax inspired by Lisp (simple & flexible)

```
(def-group «low-accounts» :parent «all»
  :select (and (= (attr «className») «Account»))
               (< (attr «balance») 300))
  :bind («logging»))
```

Context-Awareness

- Non-Functional aspects depend on the execution environment
- Simple monitoring framework
  - reifies the environment as a tree of resources
  - resources described by dynamic attributes
- Adaptation rules: `env. condition` → `action`
  - action: require a specific aspect implementation

System Policies

- DSL to define groups of adaptation rules
  - implements context-aware aspects

```
(def-policy «logging»
 (when (> (attr «/host/storage.free-space_kb») 5000)
  (ensure-attached «logging.file» ((format . xml))))
 (when
  (and
   (<= (attr «/host/storage.free-space_kb») 5000)
   (> (attr «/host/network.free-bandwidth_kbs») 10))
  (ensure-attached «logging.network»
                   ((format . text)))))
```

Putting it all together

- At startup, our system:
  - loads policies
  - sets up the monitoring framework
  - starts listening to events
- Object creation or attributes changes:
  - object moves in the most specific group
- Environment change:
  - adaptation rules re-evaluated and (de)activated
- At each moment, objects in group are weaved with all the active rules bound to the group

Conclusion

- Uses AOP paradigm:
  - base program: functional code
  - aspects: non-functional services
- Weaving specification:
  - externalized using DSLs
  - dynamic, context-aware
- Non-functional services always adapted to the state of the execution context

Perspectives

- Enable dynamic management of policies
  - loading, unloading
  - requires formalization (conflict detection…)
- Work on reconfiguration operations
  - safety guarantees, aspects composition
- Distributed behavior
  - global consistency
  - cooperation for global adaptation