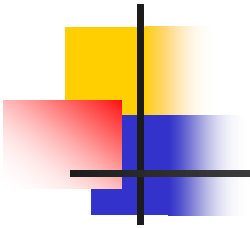
Making Aspects Natural: Events and Composition

Shmuel Katz
The Technion—Israel Institute of Technology

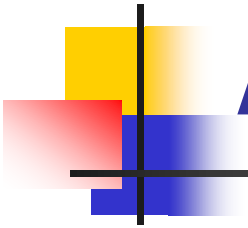
Work with: Christoph Bockisch, Somayeh Malakuti, and Mehmet Aksit
EWI, University of Twente

Or: Reinventing AOP for Complex Event Processing

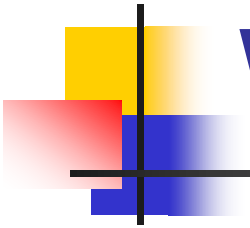


- AOP is *almost* appropriate for Complex Event Processing
- Paradigm for identifying interesting series of events and reacting appropriately
- Can combine AI ideas and exploiting partial information for Web applications
- Will discuss later…

Our proposal: Extensions for Aspect Languages

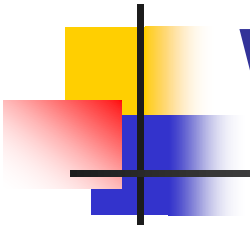


- Separate:
  - Detecting events and gathering information
  - Responding to events by changing the system or doing something externally visible
- Compose events and aspects to form hierarchies using natural terminology



Why?

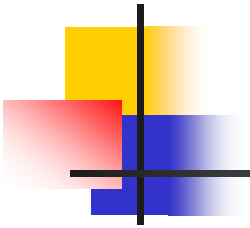
- Aspects today mix accumulating information that allows detecting events, with responding to events by making changes
  - Increases tangling and complexity
  - Decreases modularity (e.g., responding in different ways to the same event)



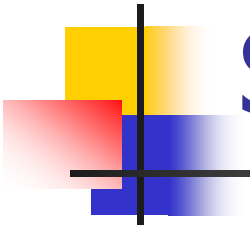
Why? (cont.)

- Aspects are too tied to code (method calls) and do not use natural terminology for the concern
  - Impedes reuse
  - Worsens fragile pointcut problem
- Combining aspects is tricky
  - Mutual influences are unclear
  - Not supported well in languages

Don't aspects already separate *when* from *what*?



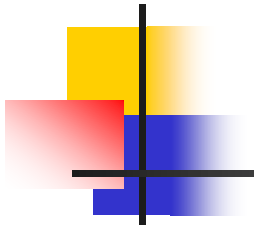
- AspectJ pointcuts versus advice
- Compose* matching patterns versus response operations in filter-types
- Problem: *when* is declarative, logic notation, but gathering information is better done interatively (in code), in conjunction with detection
- So aspect languages blur the distinction



Some other attempts

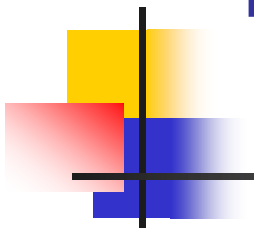
- Declarative event definition over time
  - Tracematches
  - Past-time temporal logic, regular expressions (in Compose* and temporal logic for hardware)
- Hierarchies of events
  - Abstract pointcuts
  - Jasco's separation of Hooks and Connectors
- All are partial, and still mix detection with response

Extension #1: Event declarations



- Like aspect declarations, but have no external side-effects
  - Only change locally defined fields
  - Return control to the point of activation
- Have parameter list of context exposed
- Can have trigger() operation to signal detection of the event being declared
- Support detecting and gathering info over time (a sequence of other events define this one)

Extension #2: Event detectors



- Boolean expression true when the event is detected
- Primitive expressions (like pointcut decl.)
- Names of event declarations or event detectors
- Logical combinations of detectors
- Allows having both declarative and iterative definitions of events

E-Commerce Discounts (AspectJ extension)

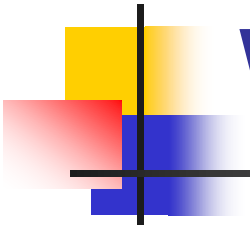
- Assume event detector called RelPur(Purchase purchase)

```
event LowActivity(P product){
    int LOWBND = 100;
    Info purchaseInfo = new Info();
    after(Purchase purchase): RelPur(purchase) {
          purchaseInfo.increase(purchase.product());
     }
    when(P product): P.timeDone() && target(product) {
           if (purchaseInfo.count(product) < LOWBND) {
                    trigger(product);
           }
          purchaseInfo.reset(product);
     }
}
```

E-Commerce Discounts (cont.)

```
event LowActPur(C cart) {
     Set<P> lowActivityProducts = new Set<P>();
    after(P product): LowActivity(product) {
                lowActivityProducts.add(product);
    }
    when(Purchase purchase): RelPur(purchase) {
        if (lowActivityProducts.contains(purchase.product())) {
          trigger(purchase.cart());
       }
    }
 }
```

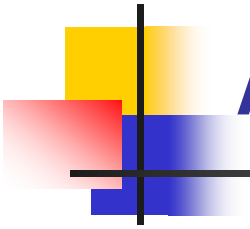
(Also has basic unit to remove products from lowActivityProducts)



Why are these events?

- Only gather information in locally defined fields
- Have no externally visible side effects (even do not print anything)
- Do not influence control
- They DO detect an event and expose relevant context, so *aspects* can react or other *events* can use them

Aspects and composition

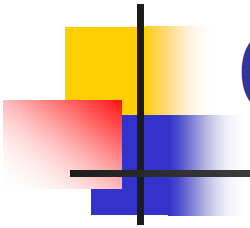


- Aspects use *basic units*: event-response pairs
- Can also compose aspects: list components and decide about mutual influences using modifiers:
  - A ignores B
  - A overrides B, A precedes B
- Modifiers are local to a composition
- Can exempt some basic units from modifiers

Aspects for discounts

```
aspect LowActDisc {
  before(C cart): LowActPur(cart) {
            cart.applydiscount(10);
   }
}
```

- Here we finally change the system---until now we just observed, and gathered information about it…
- An optimized loader would only activate the events used (directly or indirectly) in aspects (responses)



Composing aspects

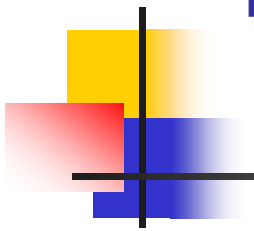
```
aspect Discount composes FreqCustDisc, LowActDisc {
  local declare overriding FreqCustDisc, LowActDisc;
}
```

- Note hierarchy of terminology, separated from the code.
- Changing the response, adding events, and making new combinations is easy, and leaves the existing events unchanged.
- If the implementation changes, only low-level events (e.g., RelPur) have to be adjusted

Development Practices Example

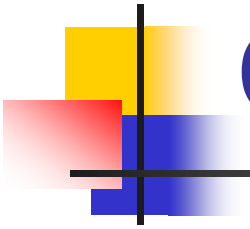
```
event TestRun(File code) :
        TestSucceeds(code) | |  TestFails(code);

event CommitWOTest(File code) {
   Set<File> testedFiles = new Set<File>();
      after(File code): Codemodified(code) {
            testedFiles.remove(code);
      }
      after(File code): TestRun(code) {
           testedFiles.add(code);
      }
      when(File code): CommitBegun(code) {
          if (!testedFiles.contains(code))
              trigger(code);
      }
 }
```

Development Practices (cont.)



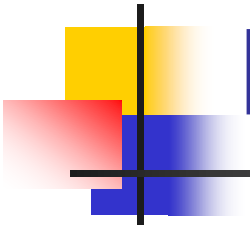
- Possible responses:
  - Log violations
  - Prevent continuing, announce problem to user
  - Try to solve it in the aspect:

```
aspect Proactive {
      void around(File code): CommitWOTest(code) {
      boolean testSucceeds = code.testSuite.run(code);
        if (testSucceeds)  proceed(code);
        else print("Correct and retest before commit");
       }
     }
```



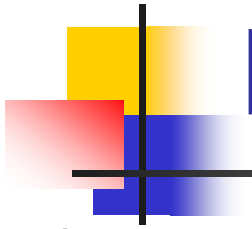
Compose* extensions

- Same basic ideas, in Compose* style.
- Have new event declaration block, and event type, that can be customized like filter types
- Events cannot modify message properties like contents, sender or receiver
- Have trigger operation in event type



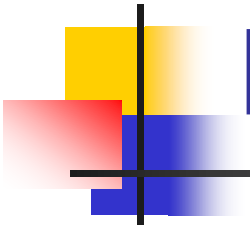
Filters and Superimpositions

- Filters (responses) now react to events, not directly to messages,
- Filters can be composed
- Superimpositions can also be composed



Runtime Verification

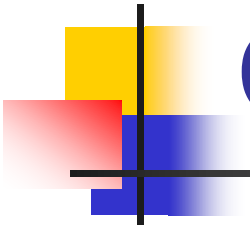
```
filtermodule ProtocolError {
 inputmessages
     ePattern:RegularExpression=
      (selector == ['read','write','open','close']) {
      event.predicate = "(open (read | write)+ close)";
       }
}

filtermodule FileErrors composes AccessError, ProtocolError {
    inputmessages
       eFileError:BooleanDetector=
          (selector==['eNotAuth', 'ePattern']);
    }
```



RV (cont.)

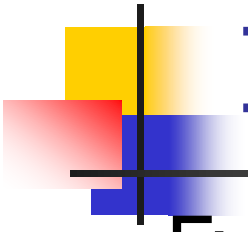
- Possible reactions:
  - Prevent access, with or without announcement
  - Log attempts at access or faulty protocol
  - Initiate special authorization interaction with user
  - Announce incursion attempt/protocol violation to manager
  - filtermodule AccessControl

```
composes PreventAccesses, LogAccesses {
 constraints
   precede (LogAccesses, PreventAccesses);
 }
```



Can we really implement?

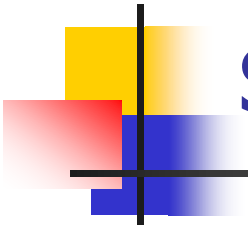
- We designed a generic prototype implementation in ALIA4J
- ALIA4J is an intermediate language for expressing complex modularity, that already is used to implement AspectJ, Compose, CaesarJ, etc.
- Provides a path to implementation, and clarifies semantic issues (see paper for details)



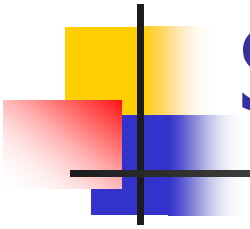
Issues(1): Order of evaluation

- Evaluate all event detectors; for all aspects having basic units with true events, do appropriate responses in some consistent order OR
- Fix a consistent order of potential aspect basic units, evaluate event detector of first and do response if it's true, move to next basic unit (interleave event detection and responses)

Issues(2): Instantiation strategy

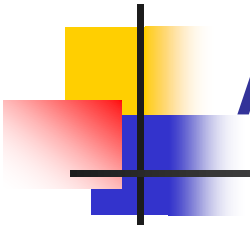


- For a system with an aspect A, and a composed aspect C made up of A and B, are there one or two versions of A?
- Each could be logical in different situations
- Could they lead to undesired changes?
- How can we control this?



Summary of the extensions

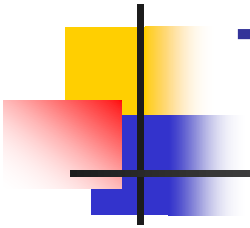
- Declare imperative event detectors to accumulate information over time
- Mix imperative and declarative detectors
- Build hierarchies of events
- Compose aspects, with modifiers to clarify relations among components
- Only aspects can have external side-effects and change the system (events cannot)



Advantages

- Cleaner modularity
- Natural expression of concerns
- Reduction in fragile pointcuts
- Greater reuse potential

- Reinventing AOP for Complex Event Processing
  - AOP: experience, compilers, and optimizations
  - CEP: clean semantics and new applications for tracking stock market, detecting incursions, finding trends, ...
  - *with* our extensions, a perfect match…



To do

- Implement using ALIA4J
- Gain further experience with examples
- Develop techniques for effective modular verification
- Consider implications for requirements and design
- Evaluate and refine
- Convince the world