Towards Right Abstraction Mechanisms for Crosscutting Concerns

Hidehiko Masuhara
University of Tokyo

東京大学
THE UNIVERSITY OF TOKYO

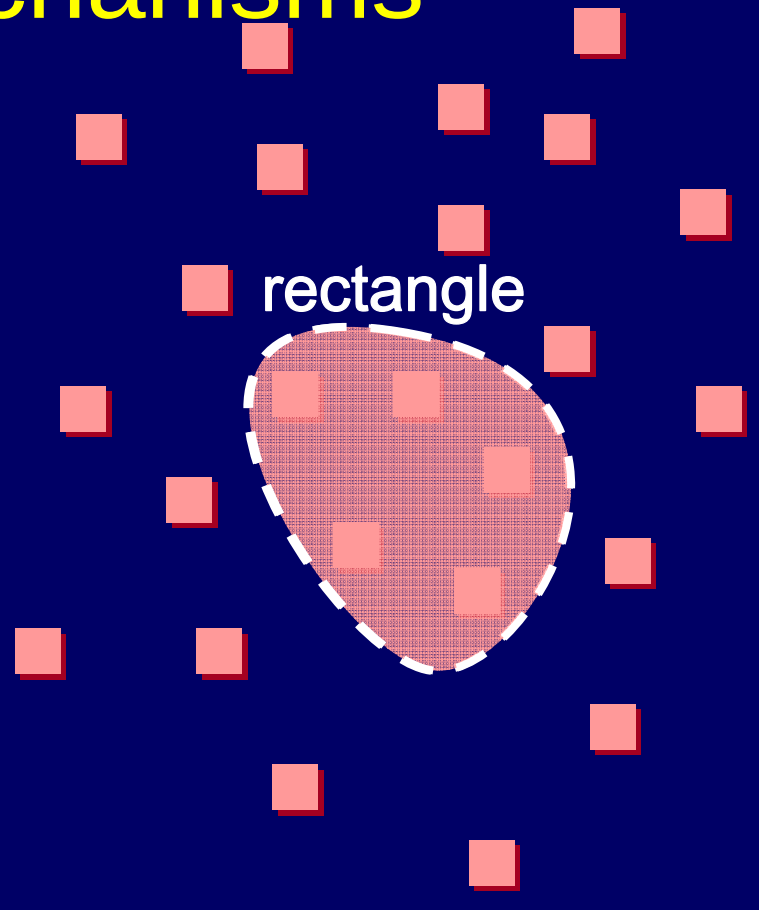
Traditional abstraction mechanisms

- Procedural abstraction
  - e.g., procedures, functions, subroutines, …
- Data abstraction
  - e.g., abstract data types
- Hierarchical abstraction
  - e.g., classes in OOP

What properties abstraction mechanisms should have?

Three properties of abstraction mechanisms

- can draw a boundary
- can name bounded entities
- can hide details



Abstraction mechanisms for crosscutting concerns?

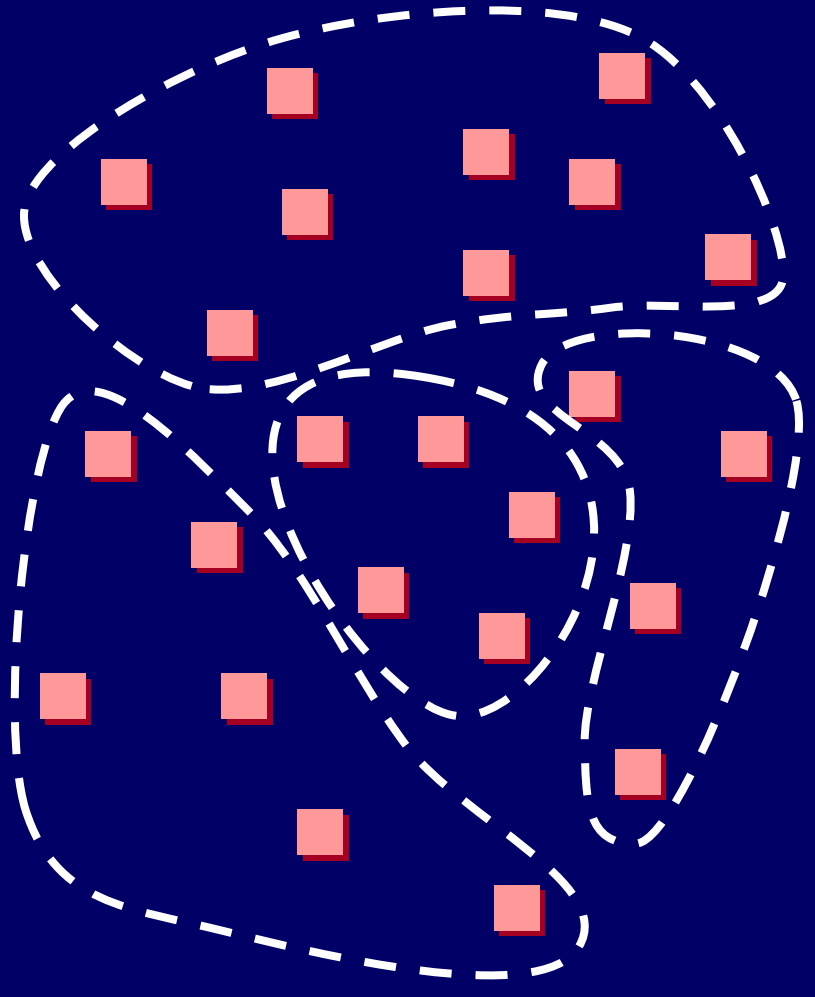
Crosscutting concerns

- Logging
- Security
- Adaptation
- Distribution
- Persistency
- Optimization
- Concurrency
- Exception handling
- …

How do you characterize crosscutting concerns (CCCs)?

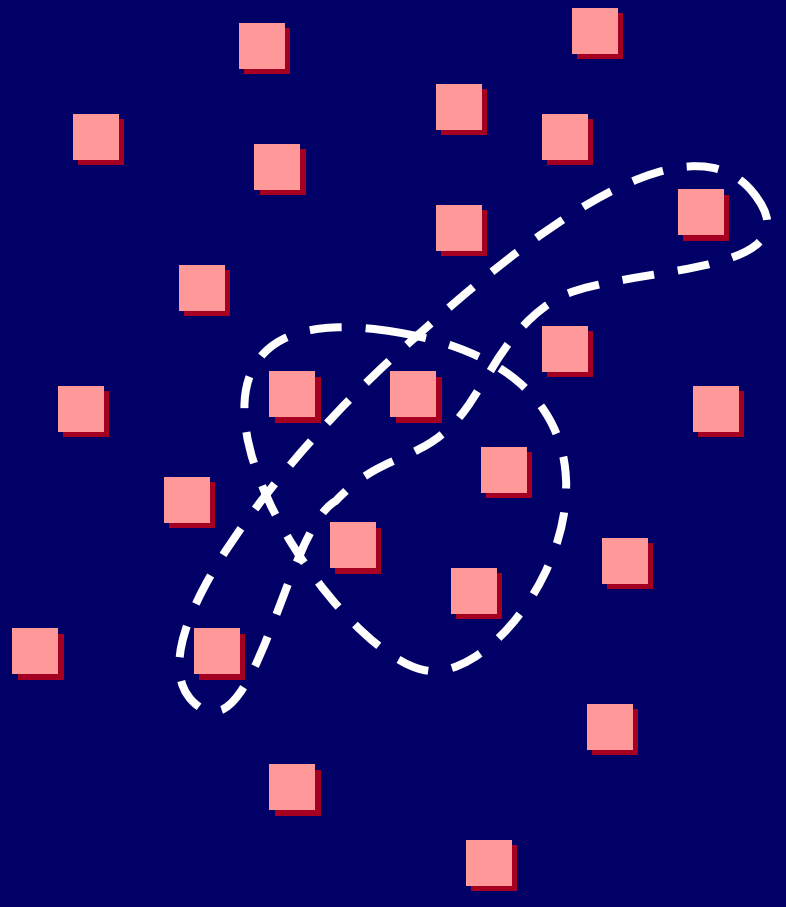
Attempts to characterize CCCs

- Those that have crosscutting structure in implementation [Kiczales91]
  - decomposition, then CCC
- A concern relating to more than one concerns
  - but what about library?
- A relationship between concerns in a crossover [ECOOP03]



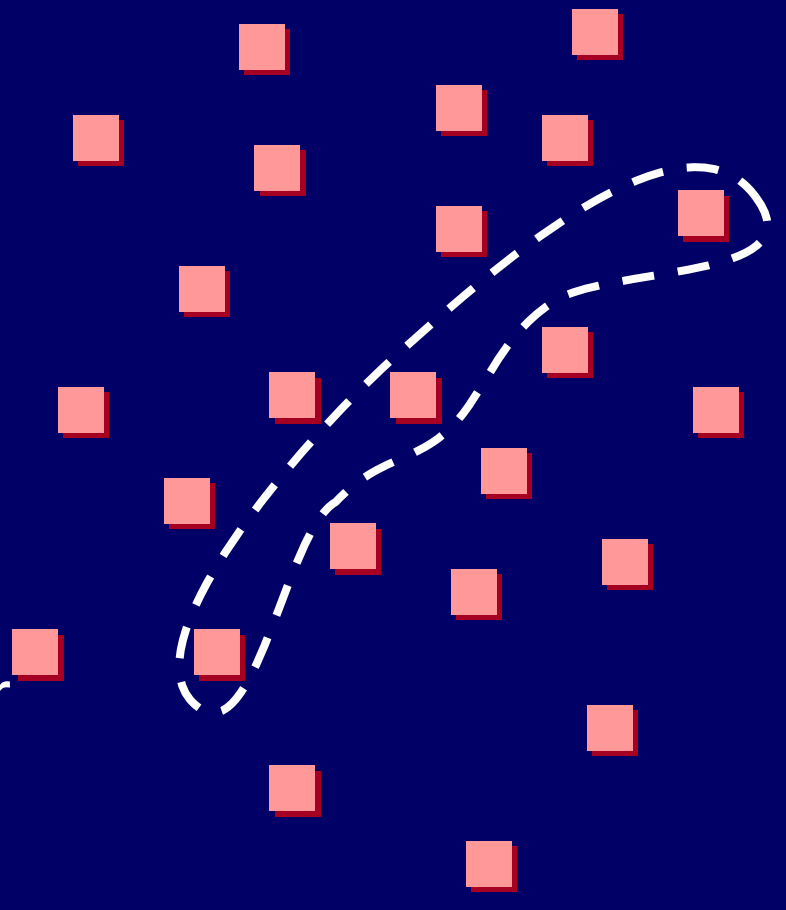
Attempts to characterize CCCs

- Those that have crosscutting structure in implementation [Kiczales91]
  - decomposition, then CCC
- A concern relating to more than one concerns
  - but what about library?
- A relationship between concerns in a crossover [ECOOP03]



CCC, in this talk

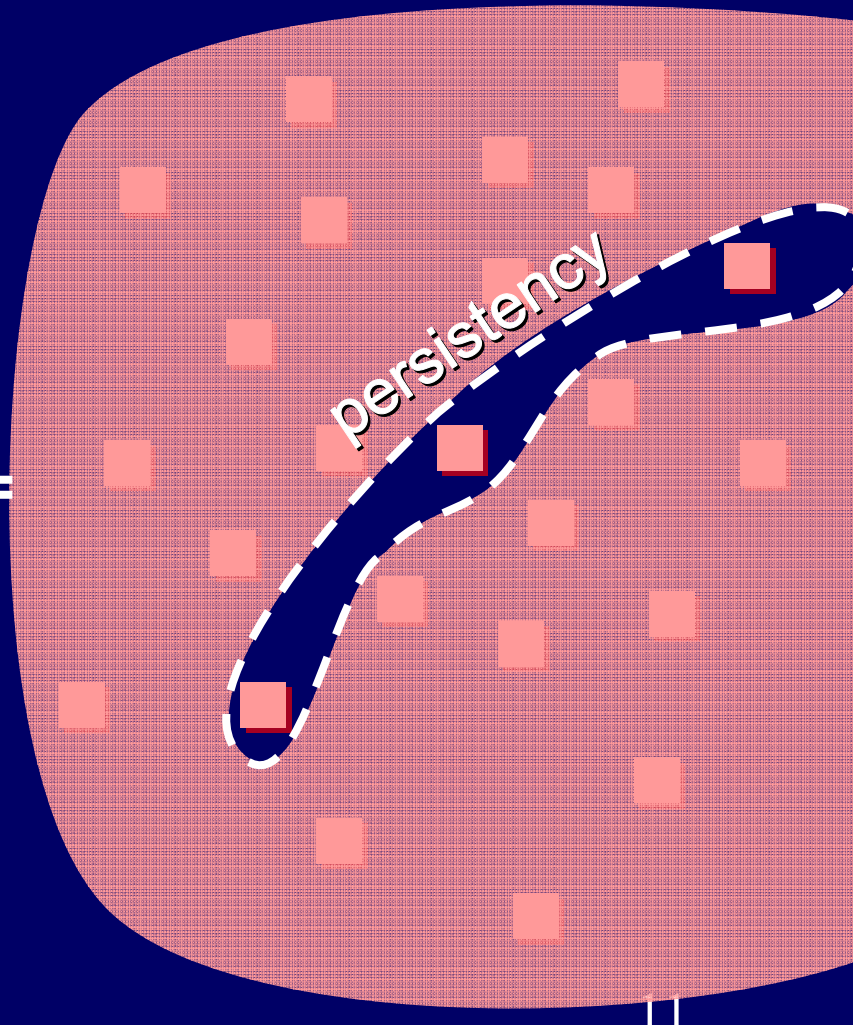
- is a concern primarily about “where to relate”
  - i.e., the shape of the boundary
  - e.g., a logging concern = “what operations we should log”
- fits Parnas’ modularization principle to hide “difficult or likely-to-change design decisions” [CACM72]



Do CCC modularization mechanisms have abstraction properties?

Three properties of ***crosscutting*** abstraction

- can draw a boundary
  - ▸ but elaborated, and
  - ▸ may not be textually structured
- can name ~~bounded entities~~ the boundary
- can hide details of ***outside*** of the boundary



Mechanisms for crosscutting abstraction

- Aspects, of course
  - pointcut and advice ⟸ focus on this
  - intertype declarations
    - let classes to implement an interface, and
    - define methods in the interface
- Layered abstractions
  - e.g., mixin layers, family polymorphism, FOP, etc.

Pointcut mechanism for drawing an elaborated boundary

- By using signatures
- By composing sub-pointcuts
- By exploiting high-level program information
  - call stack (cflow),
  - execution history (tracecut[Douence05], [Walker05], tracematch[Allan05]),
  - information flow (dflow[APLAS03]),
  - static analysis (LMP[Gybels02], Josh[Chiba04], Alpha[Ostermann05], SCoPE[AOSD07]), and so on

Pointcut mechanism for naming a boundary

- Named pointcut in AspectJ

Pointcut mechanisms for hiding details

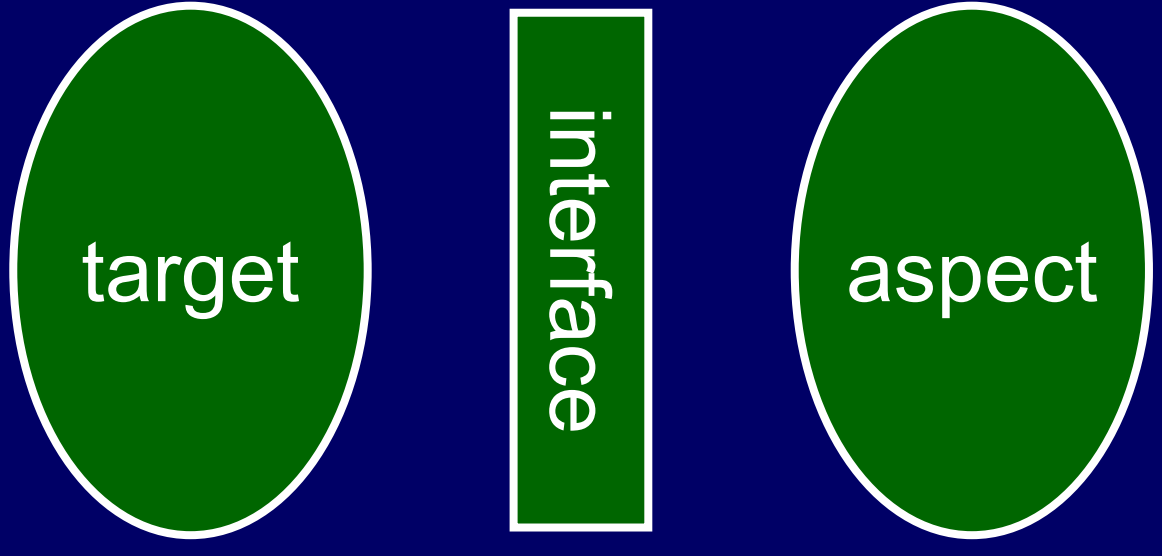
- Some hiding principles and mechanisms
  - Named pointcuts
  - Interface between target & aspect:
    XPI [Griswold06] Open Modules [Aldrich05]
- but elaboration can cause problems

Named pointcuts hide some details

- Pointcut users don't need to know parameter positions
  - pointcut dbOps(DB db):
    call(* DB.do*(..)) && target(db);
  - pointcut dbOps(DB db):
    call(* Util.db*(DB,..)) && args(db,..);

Interface between target and aspect hides details

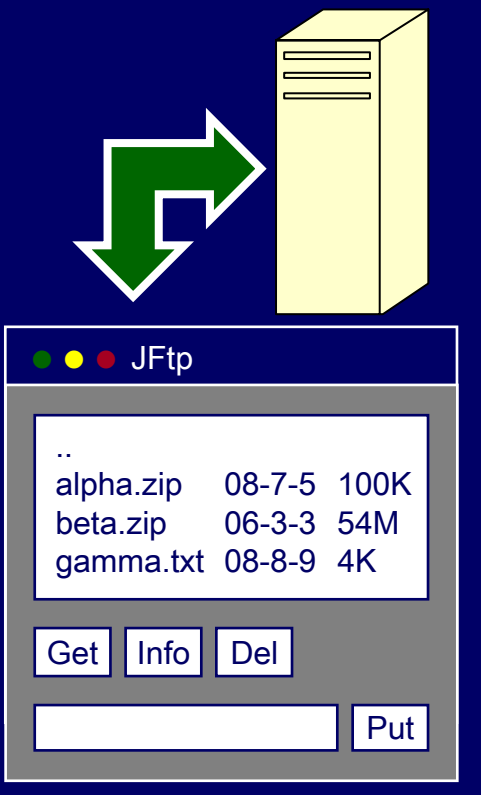
- XPI [Griswold06] and Open Modules [Aldrich05] provide separated interface between aspects and target



Elaboration can cause a problem

- Elaboration of pointcuts tend to rely on details of the target
  - see the next example…

Drawing a boundary in an FTP client



Concern: view updating
<u>when the server state changes</u>, i.e.:

- "after login, file uploading, file deletion, directory creation, directory deletion, or current directory change"
- composition mechanism helps: call(* *.doLogin(..)) || call(* *.doUpload(..)) || call(* *.doDelete(..)) …

Drawing a boundary: elaboration

Concern: view updating
when the server state changes, i.e

- "after login, file uploading, file deletion, directory creation, directory deletion, or current directory change"
- **"but only when succeeded",** because unsuccessful operations doesn't change the view
- mechanism capturing return values helps

can't do with pointcuts alone (cf. Point-in-Time JPM [APLAS06])

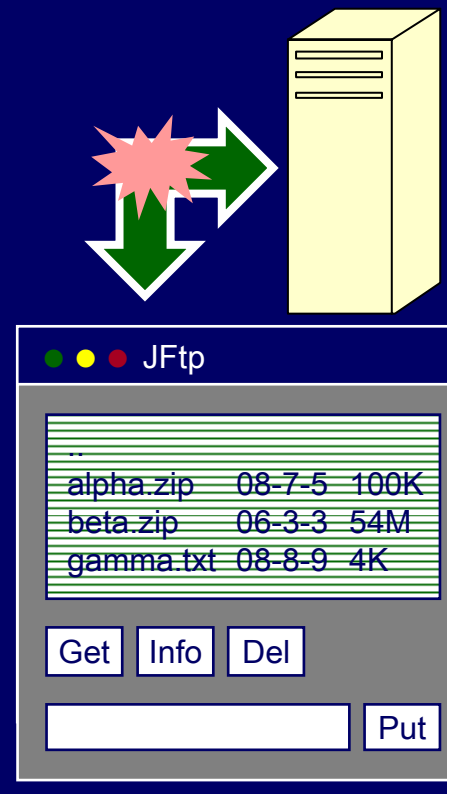
Drawing a boundary: more elaboration

- "after file uploaded, … but only when succeeded or failed due to network disconnection"
  - to make the view gray
- history-based mechanisms help

  sym send(): … sym networkError:

  sym successUpload: … sym failUpload: …

  (send* finishUpload)||(send* networkError failUpload) { … }
- ***more dependent on the details!!***



Are we doomed?

- We want an elaborated boundary
- We want to hide details

An idea to rescue: ***Example-based pointcuts***

- Instead of specifying detailed events
  - "after 1 or more sending, returned from doUpload without handling NetworkException"
- Specify by example executions, e.g., "after the program behaved like new NormalNet(). doUpload("foo") or new FaultyNet(). doUpload("foo")"
  - only depends on external interfaces

successful uploading

failure due to disconnection

Issues of providing examples

- Specifying executions
- Judging similarity of executions
- Maintaining examples

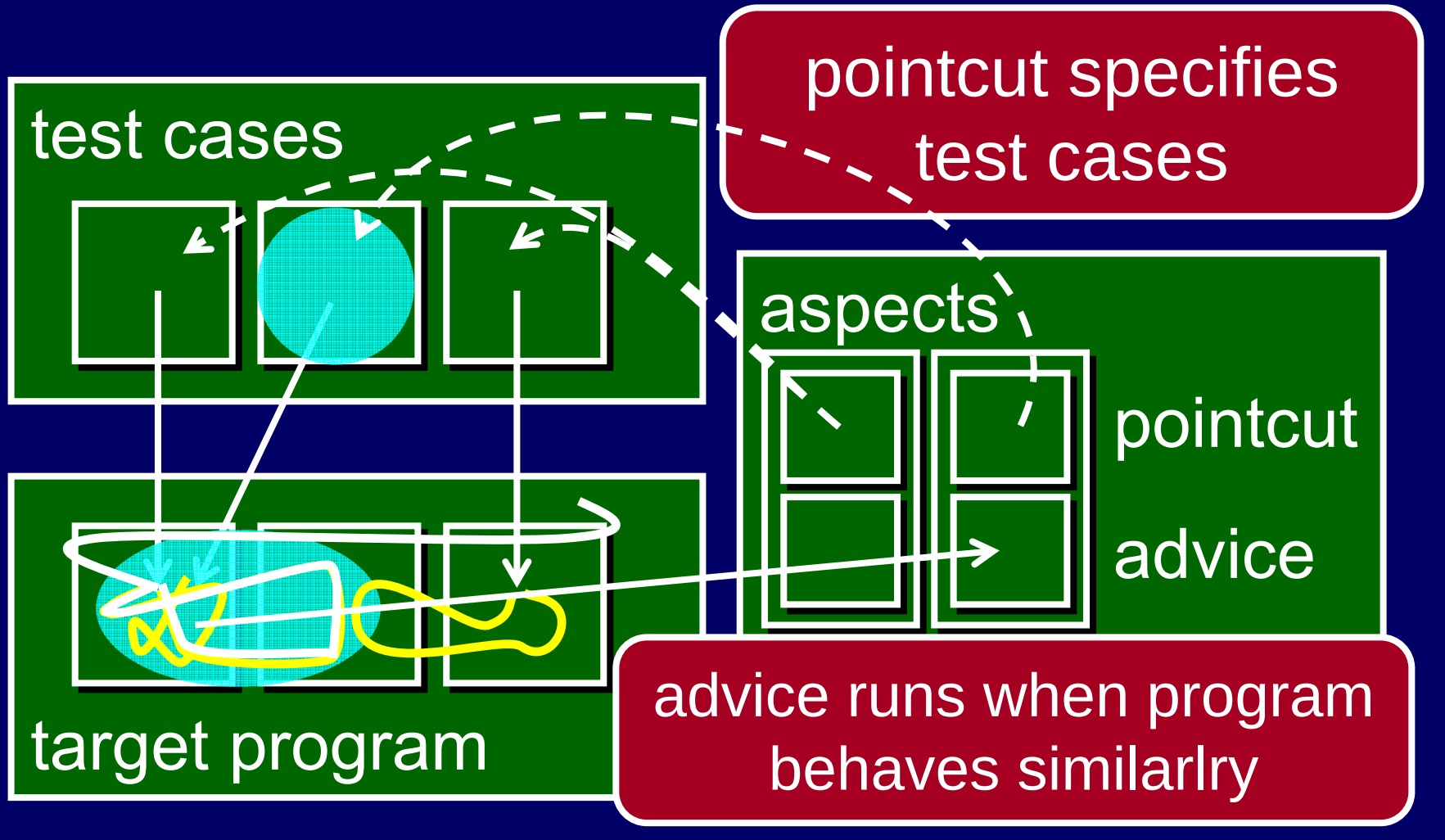
One approach:

Test-based Pointcuts

using unit test cases as examples

cf. Sakurai and Masuhara, *Test-Based Pointcuts for Robust and Fine-Grained Join Point Specification*, in AOSD'08, 2008

Test-based pointcuts: overview



Specifying executions

- Test-based pointcuts select unit test cases by specifying ***fixture variables***
  - e.g., “any unit test cases that access faultyServer”
  - can be good approximations of concerns
- requiring unit test cases to
  - define one execution per a test case
  - explicitly use fixture variables for test parameters
  - explicitly declare phases

Specifying test cases: example

```
testUploadFailureByDisconnection() {
    Server s = F.faultyServer;
    testBody(); .........
    r = s.doUpload(F.validPath);
    testCheck();
    assertFalse(r);
}
```

phase separator

F
Server normServer Server faultyServer Str vaildPath Str invalidPath

fixtures

```
after(): test(get(F.faultyServer))
            && test(get(F.validPath)) ||
         test(get(F.normServer))
            && test(get(F.validPath))
```

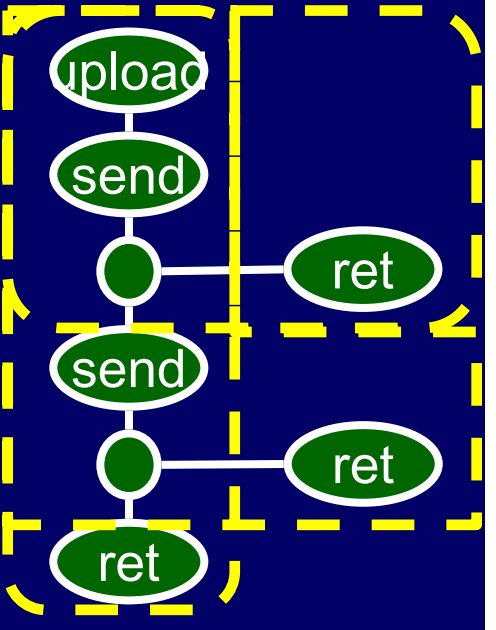
Judging similarity of executions

Candidate methods

- by entry methods — too coarse
- by execution histories
  — should distinguish # of iterations?
- by **static execution histories**
- by parameter values

Similarly wrt static execution histories

- Def. set-equality over instructions
  - ▸ includes conditional branches
- Precise enough to distinguish control-flows in a method
- Abstracting execution order / number of iterations
- Efficient implementation



Maintaining examples

Even when the target software evolves, pointcuts should be able to draw "intended" boundaries

- Test-based pointcuts can be better
  - by not directly relying on the details
  - as long as test cases are maintained — no free lunch!
  - wrt separation of responsibility

Implementation

- Prototype compiler is implemented
  - 2.5KLoC extension to abc
- 2-Phase compilation
  1. run all test cases with profiling aspects
  2. run instrumented target program
     - create a flag set at entry
     - flag at each conditional branch
     - test the falg set at exit

Challenges and other approaches to example-based pointcuts

- Test execution with/without aspects
- Ignoring unimportant control flow
  - e.g., branches to print debug messages
- Providing examples by values, or by program code
- Forward prediction
  - e.g., “when it ***will*** behave like this”

Ignoring unimportant control flow by using abstract interpretation

(suggested by Klaus Ostermann)

Abstract interpretation executes a program on an abstract domain

- e.g., D = { −, 0 , + } for integers

- Classify test parameters into “important” and “unimportant”
- Execute test programs by AI
- Ignore branches depends on “unimportant” values (and their derivations)

Examples by values

- Adaptive programming
  (e.g., Demeter / DJ)
  - ▸ focuses on the structure of values
  - ▸ based on regular expression over types
    - e.g., “from Company to Employee bypassing Customer”
- Example values can be alternatives?

Summary

- How can a “pointcut programmer” draw ***elaborated boundaries*** of join points ***with hiding details*** of join points?
- Existing mechanisms: the more elaboration, the more detail-dependent
- One approach is to use examples
  - Test-based pointcut [AOSD08]
  - Challenges and other approaches