universitätbonn ROOTS

Generic Aspects

– Enhanced Expressiveness and Reusability for AOP –

Dr. Günter Kniesel and Tobias Rho
University of Bonn

Slides on "Framed Aspects" based on
the talk at ICSR 2004 by
Neil Loughran and Awais Rashid
Lancaster University

Context

- So far we have
  - learned about the basics of aspects
  - illustrated them with the design of AspectJ
- Now we explore
  - 1. How to evolve the basic aspect concepts by making them more generic
    - ⇨ Framed aspects (preprocessors for aspects)
    - ⇨ Uniformly generic aspects (uniform language extension)
    - ⇨ This is today's topic
  - 2. How to adapt the basic aspect concepts to evolved base language features
    - ⇨ Evolution Java 2 → Java 5 reflected in AspectJ 1 → AspectJ 5
    - ⇨ This will be the topic of next week

Genericity

- A Definition
  - A language is said to support genericity if its language concepts can be parameterized by variables that represent language concepts
- An Instance: Type Genericity
  - Languages that support type genericity let type declarations be parameterized by type variables
- An Example: Type Genericity in Java

```
interface Name<TypeVar> {
        …
        TypeVar var;
        …
}
```

Genericity for Aspects

- Adaptation of type genericity from the base language
  - AspectJ 5
- External parameterization and aspect generation
  - Framed Aspects
- Pointcuts determine values for generic elements
  - LogicAJ
  - Sally
  - Carma
  - JATS
  - …

Overview of this lecture

- Framed Aspects
  - Critique of Aspects
  - Frames
  - Framed Aspects
- LogicAJ
  - Generic constraints
  - Generic advice
  - Generic introductions
- Comparison
  - Framed Aspects versus LogicAJ
  - Other Approaches

„Framed Aspects“

Genericity via Preprocessor and Explicit Enumeration of Parameter Values

Critique of AspectJ-like Aspects

Frames

Framed Aspects

Caching with AOP (using AspectJ as representative AOP technology)

```
aspect CacheAspect
  {
   private Hashtable cache = new Hashtable();

   void around(Editor g, String url): args(g,url) &&
    call(public void Network.requestInfo(Editor, String))
    {
    Document cachedPage=(Document) cache.get(url);
     if(cachedPage==null)
      {
       proceed(g,url);
       Document page = new Document(g.getDocument());
       addToCache(url,page);
      }
     else
      {
       g.setDocument(cachedPage.getContent());
      }
    }
  }
```

- Modularity preserved making it easier to evolve the caching feature
- However, …

AspectJ-like AOP is Hard to Reuse

How to store anything else than Document objects?

```
aspect CacheAspect
  {
   private Hashtable cache = new Hashtable();

   void around(Editor g, String url): args(g,url) &&
    call(public void Network.requestInfo(Editor, String))
    {
    Document cachedPage=(Document) cache.get(url);
     if(cachedPage==null)
      {
       proceed(g,url);
       Document page = new Document(g.getDocument());
       addToCache(url,page);
      }
     else
      {
       g.setDocument(cachedPage.getContent());
      }
    }
  }
```

How to operate on classes other than Network and Editor?

How to alter advice call to an alternative method?

Limitations of AspectJ-like AOP

- Difficult to adapt aspects to different contexts without affecting the code directly
  - Hard-code names of base entities
- Use of abstract aspects via inheritance may not be fine grained enough
  - No way to override advice code
- Cannot use aspect in a black box fashion
  - Need to know implementation dependencies on base code and other aspects

Merits / Demerits of AspectJ-like AOP

- Strengths
  - Advanced separation of concerns
  - Non invasiveness
  - Dynamic approaches
  - Some techniques can weave into bytecode

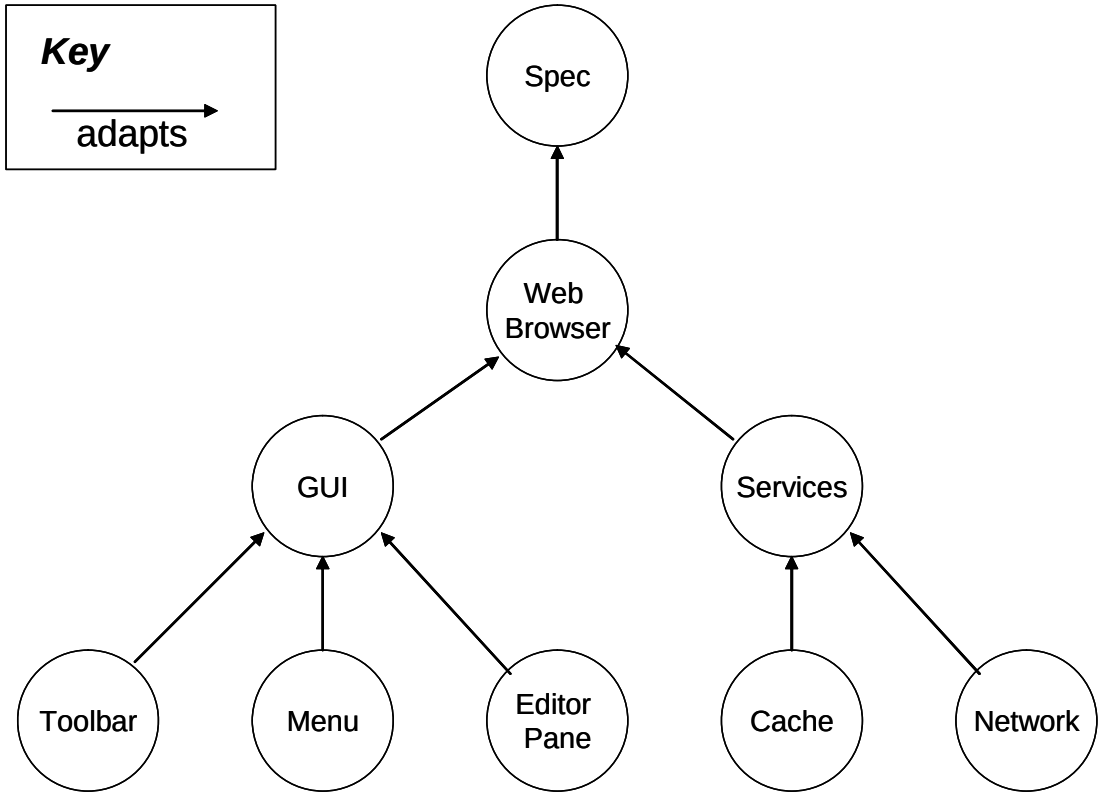
- Weaknesses
  - Limited reusability
  - No parameterisation
  - No Templates
  - No Conditional compilation

Frames

- Conceived in the 1970s by Paul Bassett
  - Adaptive reuse
  - Generative
  - Parameterisation
  - Conditional Compilation
  - Templates
- Language Independent
  - Textual documents
  - Any programming language
  - UML scripts

Frames

- Frame technology organises concerns into a hierarchy, where lower order frames refine the ones above.



Essential Frame Commands

- <set>, <set-multi>, <value-of>
  - allows frame values to be created and set
- <while>
  - creates a loop around repeating code
- <adapt>
  - instructs processor to process a frame
- <break>
  - creates a point of interest that we can add variant code to
- <option> <select> <ifdef>
  - creates and selects an option

Code Generation with Frames

Specification

```
<set multi-var = “Object” value = “String, Document, Integer, Hashtable”/>
<adapt frame = “VectorTemplate”/>
```

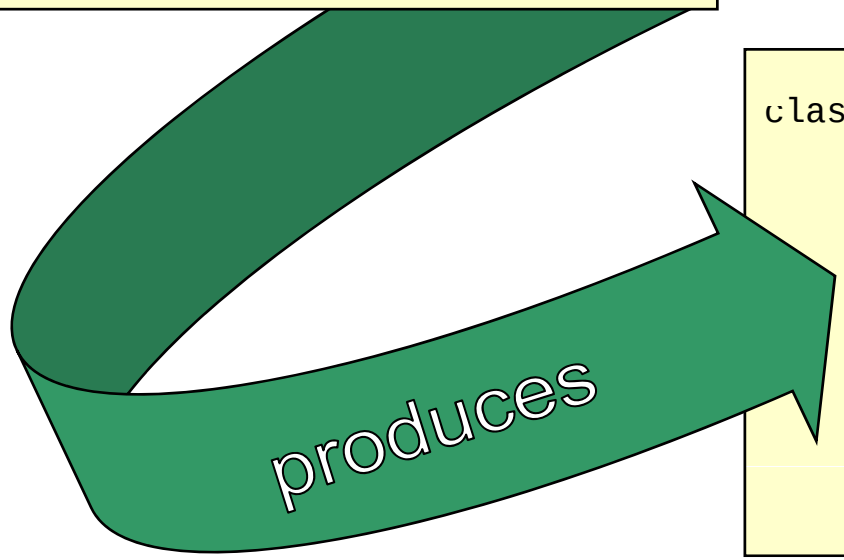
VectorTemplate

```
<while using-items in ?Object>
 class <?Object>Vector extends Vector {
    public boolean add(<?Object> o) {
      if(o instanceof <?Object>)
          return super.add(o);
      else
          return false;
    }
 }
```

Output

```
class StringVector extends Vector {
  public boolean add(String o) {
     if(o instanceof String)
         return super.add(o);
     else
         return false;

  }
}
```




```
class DocumentVector extends Vector
    {
    public boolean add(Document o)
        {
        if(o instanceof Document)
           return super.add(o);
        else
            return false;

        }
     }
```

Caching Example

- Simple Editor Pane…
  - When user clicks on a hyperlink, call the requestInfo method in the Network class

```
class Editor extends JEditorPane implements HyperlinkListener {
  private Network network;
  //.. constructor and editor initialisation
  //.. various methods

  public void hyperlinkUpdate(HyperlinkEvent e) {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED) {
      String url = e.getURL().toString();
       network.requestInfo(this, url);
    }
  }
}
```

- How does this change if we want to add caching?
  - … manually and
  - … using frames!

Editor Pane with Caching Code

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;

  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache


  //.. constructor and editor initialisation
  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();

      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {

        network.requestInfo(this, url);

        addToCache(url, this.getDocument);
        }
       else
        {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
        }

      }
    }
  }
```

Tangled Code

Editor Caching with Frames

```
class Editor extends JEditorPane implements HyperlinkListener
  {
  private Network network;
  <option cache>
  private Hashtable cache = new Hashtable();
  // .. methods for adding and retrieving data to/from cache

  </option>

  //.. constructor and editor initialisation
  public void hyperlinkUpdate(HyperlinkEvent e)
    {
    if (e.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
      {
      String url = e.getURL().toString();
      <option cache>
      Document cachedPage = (Document)getFromCache(url);
      if(cachedPage == null)
        {
      </option>
        network.requestInfo(this, url);
        <option cache>
        addToCache(url, this.getDocument);
        }
       else
        {
        // get record from cache and display it
        this.setDocument((Document)cachedPage.getContent());
        }
        </option>
      }
    }
  }
```

Variable but still Tangled Code

Frames: Strengths and Weaknesses

- Strengths
  - Easy to learn
  - Natural to use: See a variation point → tag it!
  - Language independent
  - Very flexible configuration
  - Highly reusable

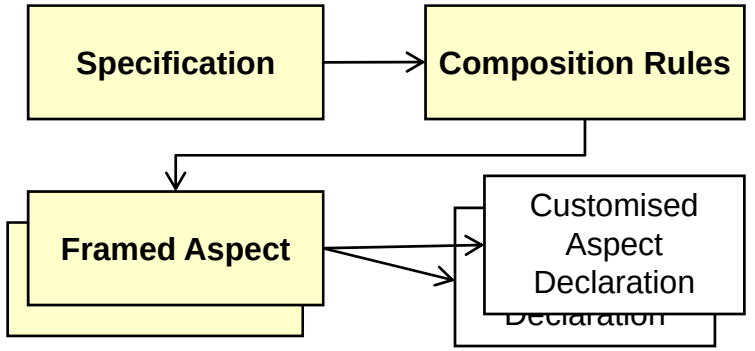
- Weaknesses
  - Ineffective when dealing with tangled and crosscutting concerns
  - Invasive approach can lead to heavily tagged code

Framed Aspects

- Combine Frames and AOP together to enhance
  - Configurability
  - Reusability
  - Modularity
  - Readability
  - Evolvability
- Basic Idea
  - Use Frames to adapt aspect code, not base code!
- Lancaster Frame Processor
  - Removal of break command to encourage modularisation via AOP advice.
  - Addition of composition rules module

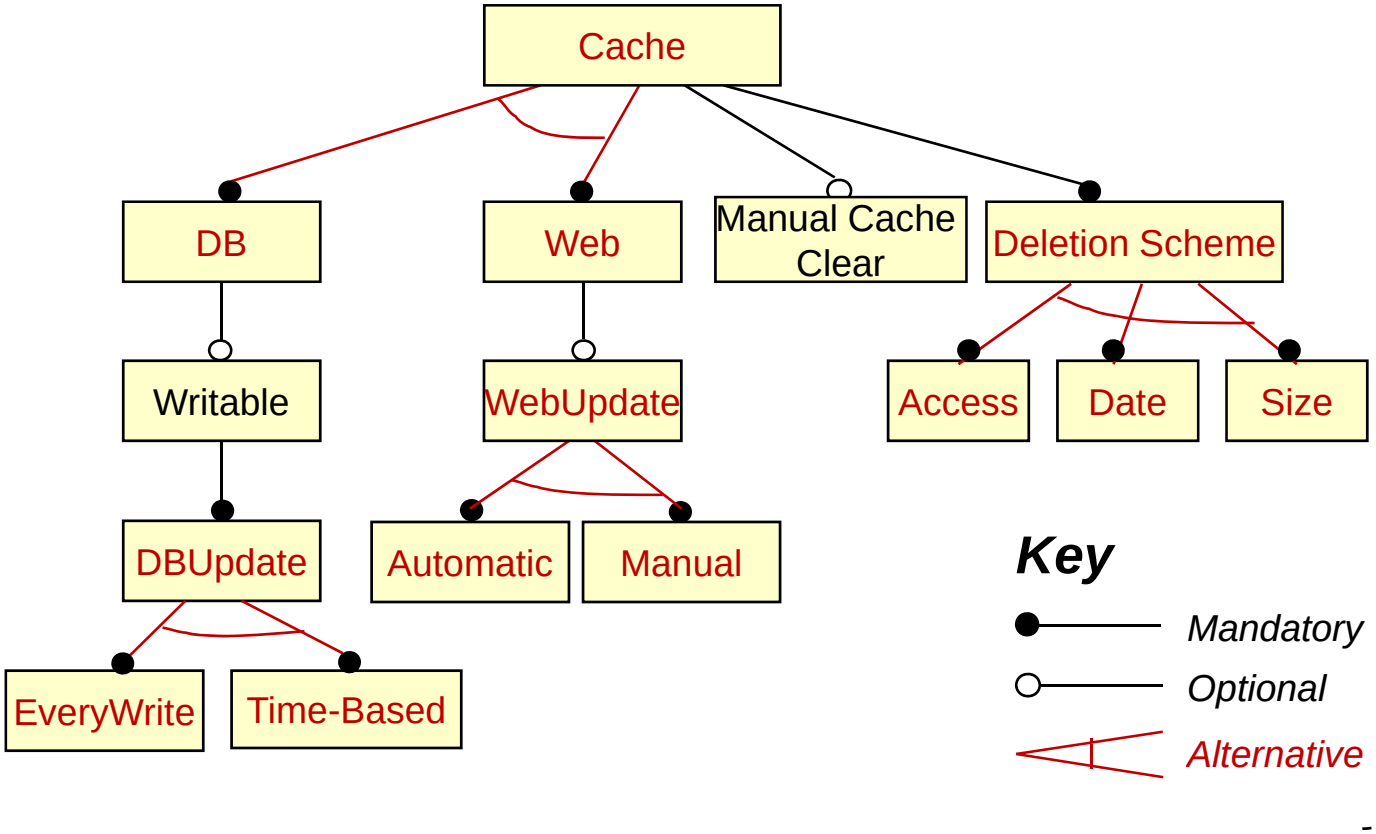
Framed Aspect Approach (Second Version)

- Made up of three elements
- Specification
  - The developers customi-sation requirements
- Composition Rules
  - Check for specification validity
  - Bind the framed aspects together
- Framed Aspects
  - Parameterised, generic aspects that can be bound to new specifications
  - Contain explicit, frame-style variation point markers



Methodology: Domain Analysis and Feature Diagrams

- Example: Reusable plug-in cache for SQL caching and Web caching



Essential assumption

Variants can be statically enumerated

Composition Rules Module

```
<constrain option = "CACHE_TYPE"         toSet = "DATABASE, WEB" />
<constrain option = "DELETION_SCHEME"    toSet = "ACCESS, DATE, SIZE" />
<constrain option = "DB_UPDATE_SCHEME"   toSet = "EVERY_WRITE, TIME_BASED" />
<constrain option = "WEB_UPDATE_SCHEME" toSet = "AUTOMATIC, MANUAL" />
<constrain var    = "PERC_TO_DEL"        toBoundary = "25,100" />

<adapt frame = "CACHE.frame" />
<adapt frame = "CACHE_TYPE“ />

<if option = "MANUAL_CACHE_CLEAR" value = "TRUE">
   <adapt frame = "MANUAL_CACHE_CLEAR.frame" />
</if>

<if option = "CACHE_TYPE" value = "DATABASE">
   <if option = "WRITABLE" value = "TRUE">
        <adapt frame = "WRITABLE.frame" />
        <adapt frame = "DB_UPDATE_SCHEME“ />
   </if>
</if>

<if option = "CACHE_TYPE" value = "WEB_CACHE">
   <if option = "WEB_UPDATE" value = "TRUE">
        <adapt frame = "WEB_UPDATE_SCHEME“ />
   </if>
</if>

<adapt frame = "DELETION_SCHEME"/>
```

Specification Module

```
<select option = "CACHE_TYPE" value = "DATABASE" />
<set var = "MAX_CACHE_SIZE" value = "1000" />

<select option = "DELETION_SCHEME" value = "ACCESS" />

<set var = "PERC_TO_DEL" value = "50" />
<set var = "CONN_CLASS" value = "DBConnection" />
<set var = "SEND_QUERY" value = "sendQuery" />
<set var = "REPLY_CLIENT" value = "replyToClient" />
<set var = "DOC_TYPE" value = "String" />

<select option = "WRITABLE" value = "TRUE" />
<select option = "DB_UPDATE_SCHEME" value = "EVERYWRITE" />

<adapt frame = "CACHE_RULES"/>
```

Current Projects

- Applied to
  - AspectJ
  - Aspectwerkz/JAC
  - AO Requirements (ARCADE)
  - Product line of existing games on J2ME enabled phones

References

- Publication
  - N. Loughran, A. Rashid: “Framed Aspects: Supporting Variability and Configurability for AOP”. *International Conference on Software Reuse* (*ICSR 2004*), Madrid, Spain.
- Downloadable version
  - http://info.comp.lancs.ac.uk/publications/Publication_Documents/2004-Loughran-Framed%20Aspects.pdf

Conclusions of the Authors

- AOP
  - lacks mechanisms for enabling reuse, making them limited in high reuse domains such as product lines.
- Frame technology
  - cannot modularise crosscutting features adequately, leading to spaghetti code and explosion of meta tags which limits understanding.
- Framed aspects
  - combine the respective strengths of AOP and frames thus providing reuse mechanisms for AOP and improving crosscutting support for frame technology.

Limitations of Framed Aspects (FA)

- Assumption of statically enumerable variations
  - Strongly restricts applicability of FA
    - ⇨ Aspect reuse only for a limited number of anticipated variations
    - ⇨ Not truely generic: Unknown variations cannot be captured
  - Makes FA highly error-prone
    - ⇨ Easy to make errors in static enumeration of legal values
    - ⇨ Especially, if only specific combinations of values vor different options are legal
  - Makes FA tedious
    - ⇨ Hard to enumerate all options, their combinations and dependencies
  - Makes FA hard to evolve
    - ⇨ Adding further variations requires extensive editing of frame composition rules and specifications

Is There More to Aspect Genericity?

Desired properties: Parameterization with

- No need to enumerate parameter values statically
- No need to write different code for each combination of parameter values
- Applicability to arbitrary, unanticipated base programs

What you want is „Uniform Genericity“!

LogicAJ: Uniform Aspect Genericity

Generic constraints → Example: Contract Enforcement

Generic advice → Example: Mock Objects

Generic introductions → Example: …

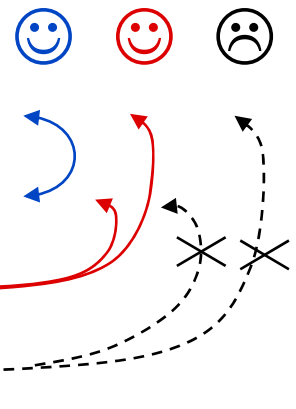
Syntax + Semantics

Contracts

- Large Software Systems rely on Contracts
- Java J2SE: “equals()-hashCode() contract”
  - equals() method and hashCode() method should always be redefined together
- Eclipse: “Internal Package Access Contract“
  - see next page

Eclipse: "Internal Package Access Contract"

- Internal package
  - indicates code to be used only by its creators
  - name pattern <superPackage>**.internal.**<arbitraryPath>.<type>
- Contract: An element of an internal package may be accessed only from
  - an internal package of its superpackage or
  - its superpackage
- Example
  - demo.internal.gui.A
  - demo.internal.core.B
  - demo.D
  - other.E



Hash-Code Contract in AspectJ?

```
declare error:
    execution(* *.equals(Object)) &&
    !existsMethod(*.ClassOfEqualsMethod.hashCode()) :
    "Contract violation!";
```

- The second condition cannot be expressed in AspectJ!
- 1) No means of testing for method existence
  - This is a static property, not a join point!
  - It is unrelated to the matched `execution` join point, so it cannot be accessed via `thisJoinPointStaticPart()`
- 2) Limitation of wildcards
  - Matches too much
  - Cannot express which wildcards must match the same value

Generic Hash-Code Contract!

```
declare error:
    execution(* ?ClassOfEqualsMethod.equals(Object)) &&
    !method(* ?ClassOfEqualsMethod.hashCode()) :
    "Contract violation!";
```

„Predicate pointcuts“ match static structure regardless of any join point

Metavariables get consistent values from matches of pointcuts (including „predicate pointcuts“)

Base Code

```
class Violation {
    public boolean equals(Object o) {
        return this == o;
    }
}
```

Generic Hash-Code Contract: Improved!

```
declare error:
    method(* ?ClassOfEqualsMethod.equals(Object)) &&
    !method(* ?ClassOfEqualsMethod.hashCode()) :
    "Contract violation!";
```

„Predicate pointcuts“
match static structure
regardless of any join point

- We want to verify the contract even for methods that are not called in the program!
- Using the method pointcut twice is what we really need in this case!

LogicAJ: Meta-variables and predicate pointcuts

Meta Variables

- Meta-Variable (MV)
  - A variable whose values are language elements
  - Types, fields, methods, field accesses, method calls, …
- Syntax
  - MV ::= ?Identifier

- List meta-variable
  - A MV whose value is a list of language elements – including the empty list
  - Parameters, arguments, …
- Syntax
  - List MV ::= ??Identifier

```
call(??Modif ?RetType ?DeclType.?Name(??Params))
 get(??Modif ?RetType ?DeclType.?Name          )
```

Meta-Variables in LogicAJ

- Range over
  - Types
  - Identifiers, Modifiers
  - Method Bodies
  - Parameters
  - Literals (Strings, Integers)
  - Lists of the previous types
- Meta-variables may appear wherever the above syntactic elements may be used in AspectJ
  - In pointcuts, introductions and advice code!
- Meta-variables in LogicAJ are logic meta-variables!

Logic Meta Variables

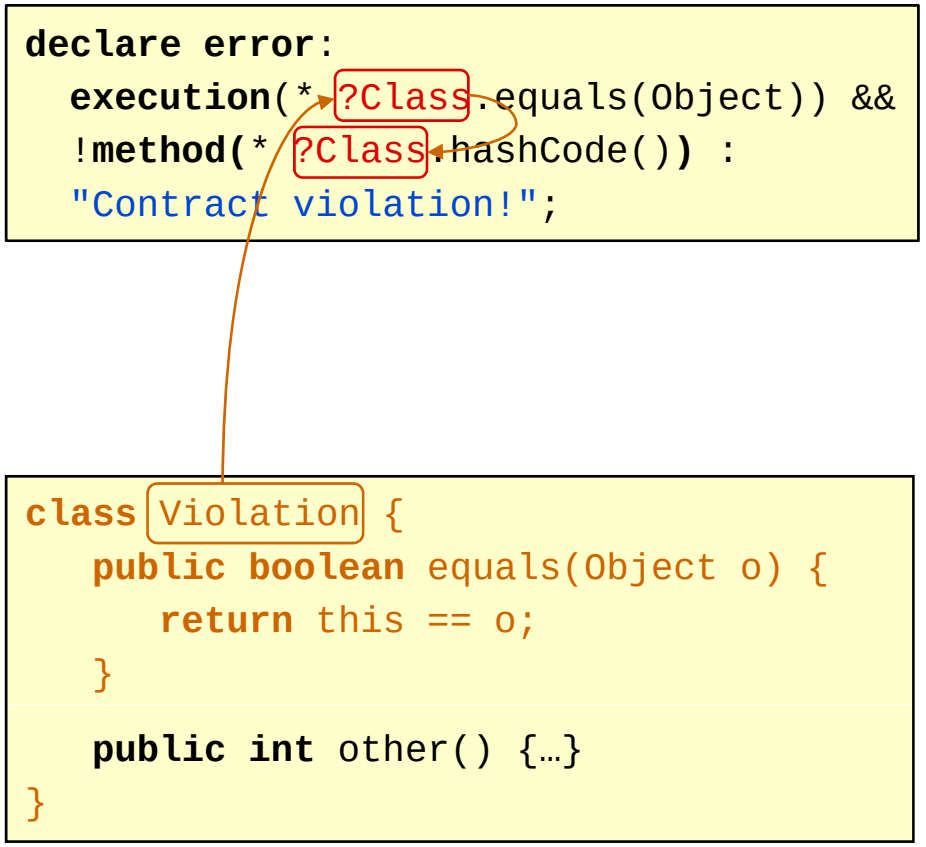
- Logic MVs / Logic list MVs
  - Their values cannot be set by assignment!
  - They get values (“bindings”) only by unification!
  - As long as it has no value it is said to be "unbound“
  - Bound logic variables are implicitly replaced by / treated as their bound value.
- Unification
  - Equality test that makes its parameters equal, if possible.
  - Can be performed implicitly by pointcut matching or explicitly via the “equals(arg1, arg2)” predicate

Explicit Unification: “equals(arg1, arg2)”

- If both arguments are constants, check whether they are equal.
  - equals(1,1) → true
  - equals(1,2) → false
- If one argument is an unbound variable bind it to the value of the other argument and return true.
  - equals(?Class,“Array”) → true with ?Class bound to “Array”
  - equals(“Array”,?Class) → true with ?Class bound to “Array”
- If both arguments are unbound variables bind one to the other, constraining their future values to be equal.
  - equals(?Class,?Type) → true with ?Class bound to ?Type

Implicit Unification: Pointcut Evaluation

- Evaluation of a pointcut implicitly unifies each contained logic variable to the respective part of the matched program element's context
  - See "?Class" and "Violation"
- Different occurrences of the same variable must be unified to the same value. Otherwise the pointcut does not match!
  - See the two occurrences of "?Class"
- A logic variable in a negated pointcut must be bound by a previous positive occurrence of that variable
  - See "!method(?Class…)"



Predicate Pointcuts

Primitve Pointcuts

- bind join point (shadow) to implicit logic variable "?jp"
- bind explicit logic variables to parts of the join point shadow's context

call(* *.?m)

join point shadows and their context

Predicate Pointcuts

- do not bind join point
- bind explicit logic variables to any static program elements and constants

method(* ?type.?m())

arbitrary static program elements

{{ *?jp* = o.method1(), ?m = method1},
{ *?jp* = this.method1(), ?m = method1},
{ *?jp* = this.method2(), ?m = method2}
}

{ { ?type = Class1, ?m = method1},
{ ?type = Class2, ?m = method2} ,
{ ?type = Class3, ?m = method3}
}

Predicate Pointcuts

- Static structure queries
  - **method**(*Modifier RetType DeclType.Name*(*Params*))
    - ⇨ `method(public void Target.?m(?params))`
  - **field**(*Modifier RetType DeclType.Name*)
    - ⇨ `field(public int Target.?f)`
  - **subtype**(?type, ?subtype)
    - ⇨ `subtype(TreeObject, ClassDef)`
- Unification
  - **equals**(?fn1,?fn2)
    - ⇨ `equals(...,...)`
- List handling predicates
  - **member**(Elem,List), **removeElement**(Elem,L1,L2)
  - **head**(Head,List), **tail**(Tail,List), …
- String handling predicates
  - **concat**(S1,S2,S12), **pattern**(Pat,Vars,Vals), …

See http://roots.iai.uni-bonn.de/research/logicaj/documentation/pointcuts for a complete list

Explicit Joint Point Variable

- Sometimes it is necessary to express that two pointcuts do not refer to the same join point
  - Example: Implementation of the equals / hashCode contract using only execution pointcut:

```
declare error:
   execution(* ?Type.equals(Object)) &&
   !execution(?other, * ?Type.hashCode()) :
   "Contract violation!";
```

Makes join point variable explicit and names it differently to express that its value may be different from the implicit join point "?jp"

- General syntax
  - <primitive pointcut name>**(**?mv**,** <pointcut pattern>**)**
  - E.g. **`call(`**?call_jp**,** ?ret ?type.m()**)**

Generic Pointcut Semantics (1)

- A binding is an association of a logic variable to a constant, a program element or a logic variable
  - ?X ←1
  - ?type ← Class1
  - ?Y ← ?X
- A substitution is a set of bindings for different logic variables
  - {?X ← 1, ?Y ← ?X}
- A ground substitution is a substitution that binds all variables directly or indirectly to a constant or program element
  - {?X ← 1, ?Y ← ?X} is a ground substitution for an expression that only contains the variables ?X and ?Y

Generic Pointcut Semantics (2)

- A substitution for a pointcut is a set of bindings for each of its logic variables, including the implicit ?jp variable
  - {?jp ← o.method1(),
    ?type ← Class,
    ?m ← method1}
- The semantics of a generic pointcut GPC with respect to a program P is the set of all ground substitutions for GPC that make GPC true for P

```
pointcut GPC :
    call(* *.?m) &&
    method(* ?type.?m())
```

Program P

{ {?jp ← o.m1(), ?type ← Class1, ?m ← m1} ,
{?jp ← this.m2(), ?type ← Class1, ?m ← m1} ,
{?jp ← this.m2(), ?type ← Class2, ?m ← m2}
}

Summary "Generic Pointcuts"

- Generic pointcuts replace wildcards by named logic meta-variables
- The semantics of a generic pointcut is the set of all ground substitutions that make it true for a program
- Next: Generic pointcuts are the "heart and soul" of generic advice and generic introductions

Generic Advice

Motivating example: Mock objects

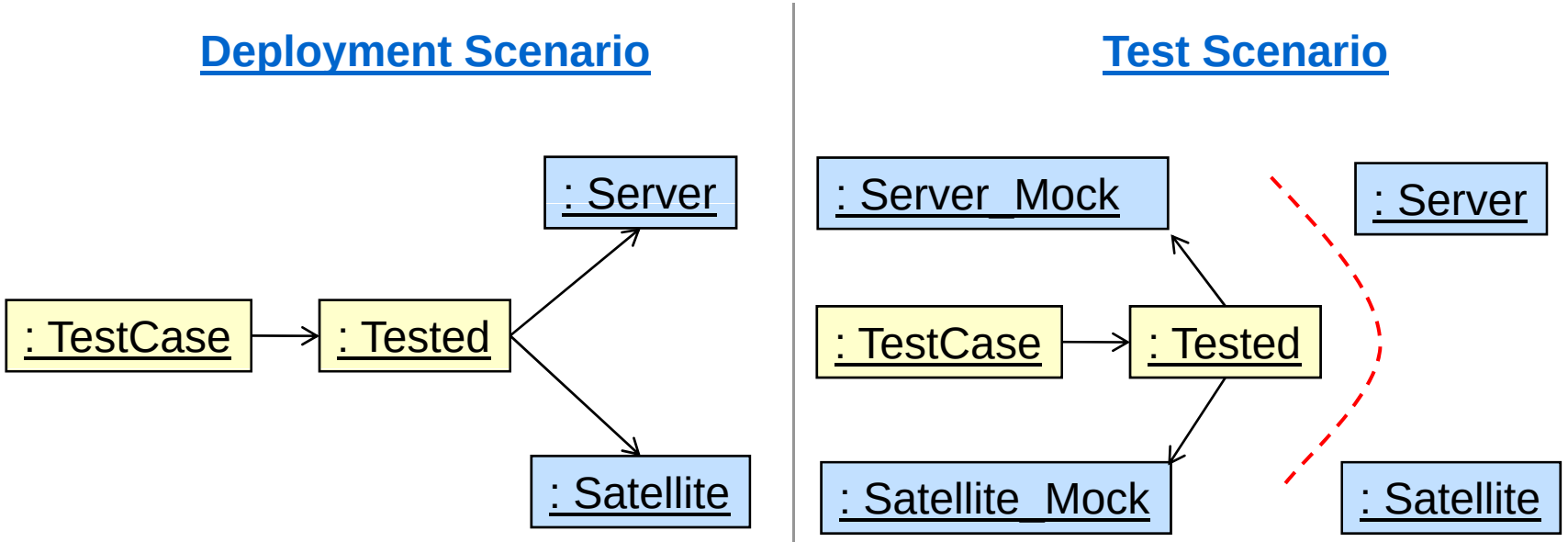
Mock objects with AspectJ (without reflection)

Mock objects with AspectJ and reflection

Mock objects with generic advice

Unit Testing with "Mock-Objects"

- Class A_mock implements correct behavior of class A
  - Just as much as necessary for a specific test case
- During testing, A instances are replaced by A_mock instances
  - Simulate behaviour of objects that are unavailable or too expensive to create or access
  - Isolate cause of failures → Tested object, not its helpers



Problem: Context Dependent Effects

- Scenario
  - New subclass should be used consistently instead of its superclass
- Challenge
  - Replace superclass constructor calls by calls of subclass constructors: `new Super(args)` → `new Sub(args)`
  - In Objective C this is built-in: „Class posing“ does just that.
  - In other language it is tedious and highly crosscutting
- Can we solve this generically with aspects?
  - Many constructors
  - Different signatures
  - Constructor names depend on the replaced classes

Class Posing via Aspect Code Bloat

- Example: Replace superclass S by new class C
  - Superclass constructors: S(String), S(String,int), ...
  - Subclass constructors: C(String), C(String,int), ...

```
aspect Replace_S_by_C {

 S around(String arg1) :
   call(S.new(..)) && args(arg1)
   {
    return new C(arg1);
   }

 S around(String arg1, int arg2) :
   call(S.new(..))&& args(arg1,arg2)
   {
    return new C(arg1,arg2);
   }

  // Continue like this ...
}
```

- Redundancy
  - New advice for every constructor
  - Similar structure
- No Reusability
  - Need to write another aspect to replace another pair of classes
  - S → C1, S → C2, ...
  - S1 → C, S1 → C1, ...

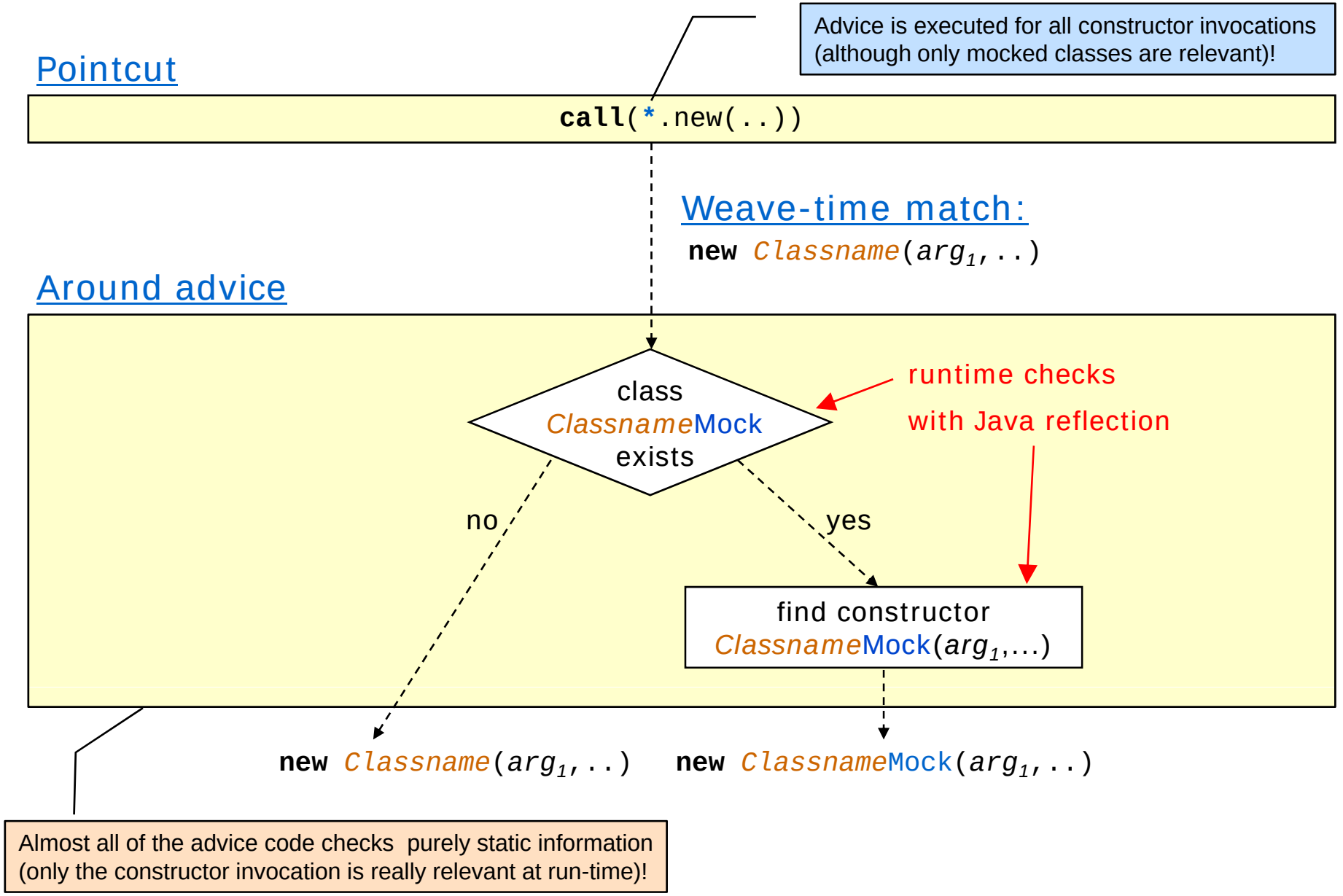
Class Posing via AspectJ & Reflection

- Example: Replace superclass S by new class C
  - Superclass constructors: S(String), S(String,int), ...
  - Subclass constructors: C(String), C(String,int), ...

```
aspect AJClassPosingWithReflection {
 abstract String getOldClassName();
 abstract String getNewClassName();
 Object around() : call(*.new(..)) {
    Class class =  thisJoinpoint.getSignature().getDeclaringType();
    Class newClass;
    if (!class.getName().equals(getOldClassName()){
        return proceed();
    }
    try {
       newClass = Class.forName(getNewClassName());
    } catch (Exception ex) {
       return proceed();
    }
    Object[] args = thisJoinPoint.getArgs();
    try {
       Class[] types =
             ((CodeSignature)thisJoinPoint().getSignature()).
               getParamtypes();
       Constructor constr = resolveConstructor(newClass, types);
       return constr.newInstance(args);
    } catch (Exception ex) {
       throw new RuntimeException("Instantiation failed");
    }
 }
}
```

- Verbose
  - 30-line reflective helper method not even shown
- Obscure
  - What's going on here?
- No static safety
  - Run time errors if class / constructor unavailable
- Inefficient
  - Intercepts all constructors
  - Checks purely static info

AspectJ Implementation



Mock-Objects in AspectJ: Code Excerpt

Advice is executed for all constructor invocations (although only mocked classes are relevant)!

```
aspect MockAspect {

  Object around() : call(*.new(..)) {
    Class class =  thisJoinPoint.getSignature().getDeclaringType();
    String name = class.getName();
    Class mock;
    try {
       mock = Class.forName(name + "Mock");
    } catch (Exception ex) {
        proceed();
    }
    Object[] args = thisJoinPoint.getArgs();
    try {
      Constructor constr = resolveConstructor(class, args);
      return constr.newInstance(args);
    } catch (Exception ex) {
        throw new RuntimeException("Instance creation failed");
    }
  }
}
```

Almost all of the advice code checks purely static information (only the return statement is really relevant at run-time)!

Mock-Objects – Concrete Example

- Replace User with UserMock
- Replace UserManager with UserManagerMock

```
...
use = new User ();
...
```

```
...
user = new UserMock();
...
```

```
...
userMng = new UserManager();
...
```

```
...
userMng = new UserManagerMock();
...
```

Mock-Objects – Generic Solution

Select all constructor calls and their arguments

Aspect

```
aspect MockAspect {

   Object around useMock(?mock, ??args) :
      call(?class.new(..)) && args(??args)
      concat(?class,"Mock",?Mock) &&
      type(?Mock) {
   return new ?Mock(??args);
   }
}
```

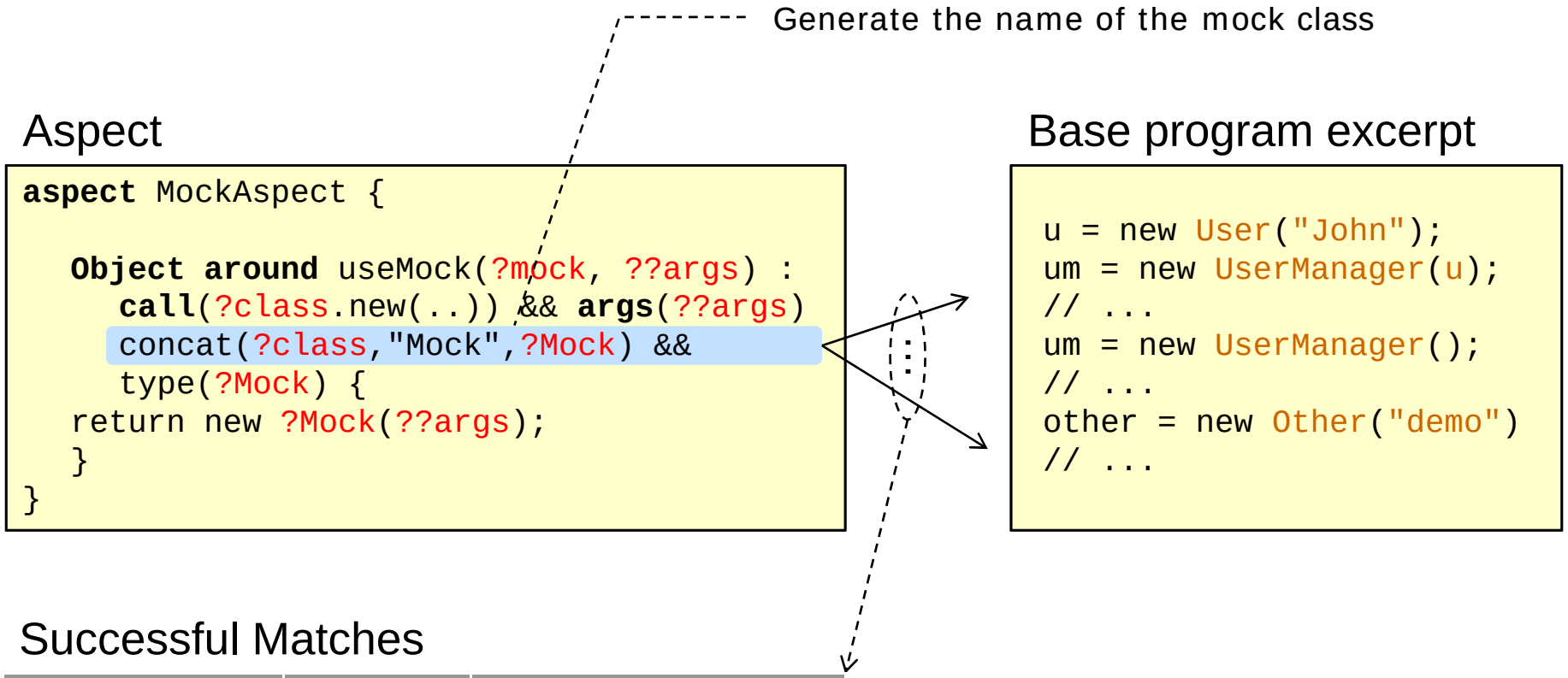
Base program excerpt

```
u = new User("John");
um = new UserManager(u);
// ...
um = new UserManager();
// ...
other = new Other("demo")
// ...
```

Successful Matches

?class	??args	?mock
User	["John"]	--- yet unbound ---
UserManager	[u]	--- yet unbound ---
UserManager	[]	--- yet unbound ---
Other	[„demo“]	--- yet unbound ---

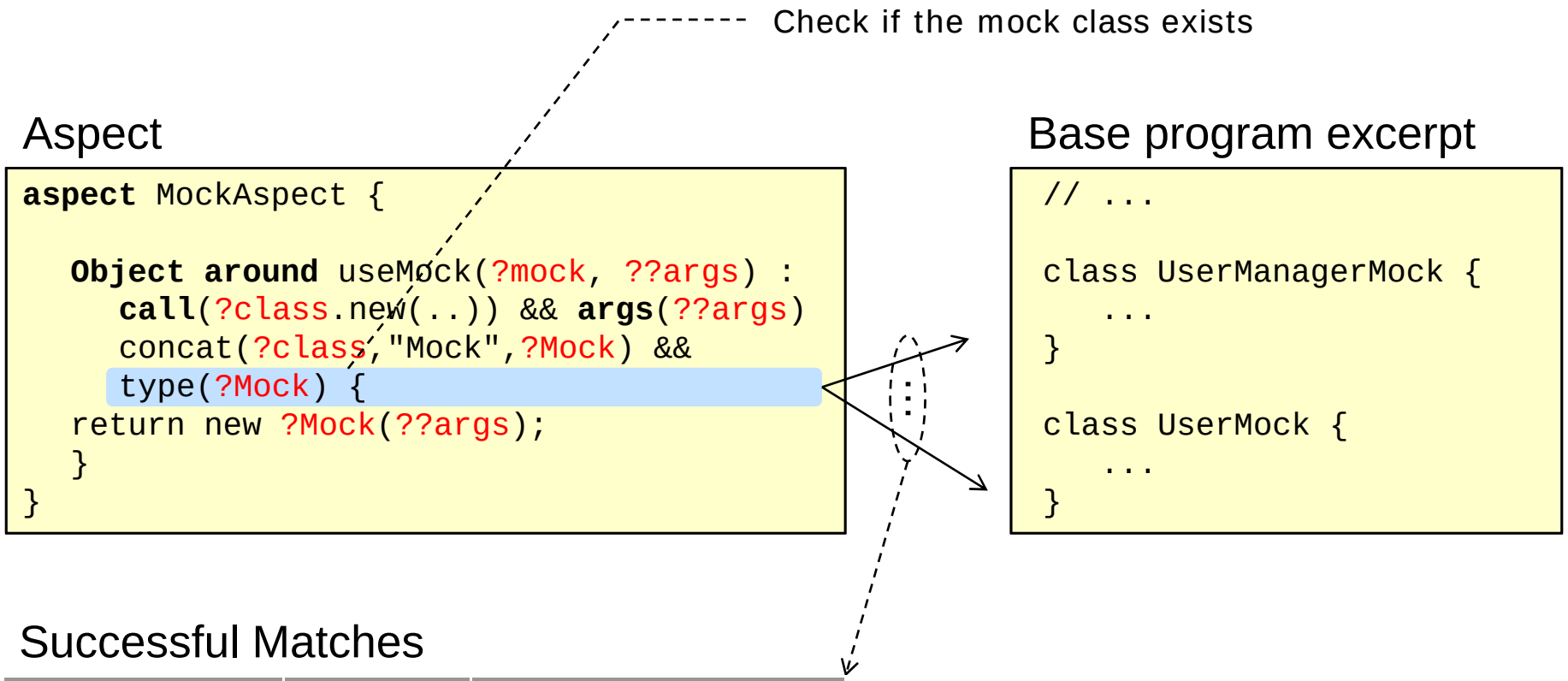
Mock-Objects – Generic Solution



Successful Matches

?class	??args	?mock
User	["John"]	UserMock
UserManager	[u]	UserManagerMock
UserManager	[]	UserManagerMock
Other	[„demo“]	OtherMock

Mock-Objects – Generic Solution



Successful Matches

?class	??args	?mock
User	["John"]	UserMock
UserManager	[u]	UserManagerMock
UserManager	[]	UserManagerMock
~~Other~~	~~[„demo“]~~	~~OtherMock~~

Mock-Objects – Generic Solution

Create instance of mock class with original arguments

Aspect

```
aspect MockAspect {

   Object around useMock(?mock, ??args) :
      call(?class.new(..)) && args(??args)
      concat(?class,"Mock",?Mock) &&
      type(?Mock) {
   return new ?Mock(??args);
   }
}
```

Base program excerpt

```
u = new User("John");
um = new UserManager(u);
// ...
um = new UserManager();
// ...
other = new Other("demo")
// ...
```

Successful Matches

?class	??args	?mock
User	["John"]	UserMock
UserManager	[u]	UserManagerMock
UserManager	[]	UserManagerMock

Run-time actions

```
new UserMock("John")
new UserManagerMock(u)
new UserManagerMock()
```

The Power of Generic Advice – Context-Dependent Aspect Effects

Aspect with Generic Advice

```
aspect MockAspect {

   Object around useMock(?mock, ??args) :
      call(?class.new(..)) && args(??args)
      concat(?class,"Mock",?Mock) &&
      type(?Mock) {
   return new ?Mock(??args);
   }
}
```

- Metavariables bound in the pointcut influence the advice
- The advice has a different effect depending on the join point context

Base program excerpt

```
u = new User("John");
um = new UserManager(u);
// ...
um = new UserManager();
// ...
other = new Other("demo")
// ...
```

Run-time actions

```
new UserMock("John")
new UserManagerMock(u)
new UserManagerMock()
```

Context-dependent Aspect Effects

Objects

- Static binding
  - **static variable type** determines behaviour
  - fixed, rigid behaviour
- Dynamic binding
  - **dynamic receiver type** determines behaviour
  - context-dependent message behaviour

Aspects

- Homogeneous Aspects
  - executed **advice** determines behaviour
  - fixed, rigid behaviour
- Heterogeneous Aspects
  - **join point context** and **advice** determine behaviour
  - context-dependent aspect behaviour

Methodology: Factor out Generic Part

```
abstract aspect ClassPosing {

    // Tell us in which types to replace constructors:
    abstract pointcut replace(?super, /*by*/ ?sub);

    // Replace constructors calls:
    Object around useSubclssInstance(?sub, ??args):
        replace(?super,?sub) &&   // Replace what?
        subtype(?sub,?super) &&   // Is it legal?
        call(?super.new(..)) &&   // Intercept ?super constructor
        args(??args)              // … and all its arguments.
    {
        return new ?sub(??args);  // Return instance of ?sub
    }
```

```
aspect YourMockObject extends ClassPosing {
```

```
aspect MyMockObject extends ClassPosing {

    replace(?super, /*by*/ ?sub) :
        // defined by enumeration or
        // arbitrary condition
    ;
```

Generic Advice Syntax and Semantics

- Generic Advice contains meta-variables in its pointcut, parameter list and body.
  - The parameter list includes the implicit "join point" meta-variable
- The body substitutions are the set of successful substitutions for the pointcut projected onto the meta-variables in the parameter list
  - The projection of { {?jp←o.m1(), ?type←Class1, ?m←m1},
    {?jp←o.m1(), ?type←Class1, ?m←m2} }
    onto ?jp, ?type is { {?jp←o.m1(), ?type←Class1} }
  - Note that projection can eliminate substitutions!
- The advice body is executed for each body substitution

Generic Introductions

Pointcuts for Inter-Type-Declarations

Singleton Pattern Example

Context-Dependent Inter-Type-Declarations?

- Problem in AspectJ
  - Inter-Type-Declarations have no pointcuts.
  - Introducing meta-variables in advice bodies only (as in "Framed Aspects") only allows anticipated, static enumeration of values.
- LogicAJ Approach
  - Inter-Type-Declarations have pointcuts!
  - Context-dependent behaviour for all aspect language elements (advice, introductions, declarations)
  - Uniform language design!

Singleton-Pattern: Introductions

```
public abstract aspect SingletonProtocol {

  /* Which class should be a singleton? */
  abstract pointcut classToSingleton( ?c2s );

  /* Introduce the static field that holds the unique instance */
  introduce singletonField( ?c2s ):
    classToSingleton( ?c2s )
  {
    private static ?c2s ?c2s._unique;
  }

  /* Introduce the method for getting the unique instance. */
  introduce singletonGetter( ?c2s ):
    classToSingleton( ?c2s  )
  {
    public static ?c2s ?c2s.getInstance() {
      if ( _unique == null ) _unique = new ?c2s();
      return _unique;
    }
  }

  // ... Continued on next slide ...
}
```

Singleton-Pattern (Continued)

```
public abstract aspect SingletonProtocol {

  // ... see previous slide ...

  /* Reroute constructor invocations to the getInstance method */
  ?c2s around useSingleton( ?c2s ) :
     call( ?c2s.new(..) ) &&
     classToSingleton( ?c2s ) &&
     args( ??args )
  {
     return ?c2s.getInstance( ??args );
  }

}
```

- Note the precise return type "**`?c2s`**" where AspectJ 1.x would require the type "**`Object`**".
- Uniform genericity subsumes type genericity!

Semantics of Introductions

- Same as for advice
  - All successful substitutions "produced" by the pointcut for the meta-variable parameters are applied to the introduction body.
  - A generic introductions whose pointcut yields N sucessful substitutions triggers N different concrete introductions.

LogicAJ Summary

Declaration, Introduction, Advice

- In LogicAJ all aspect language elements have a name and are guarded by a pointcut

declare error	*name*		**:**	*pointcut*	**:**	*stringLiteral* **;**
declare warning	*name*		**:**	*pointcut*	**:**	*stringLiteral* **;**
declare parent	*name*		**:**	*pointcut*	**:**	*interface* **;**
introduce	*name*	**(** *params* **)**	**:**	*pointcut*	**{**	*introBody* **}**
adviceKind	*name*	**(** *params* **)** **throwing** **(** *ex* **)**	**:**	*pointcut*	**{**	*adviceBody* **}**

before | **after** | **around**

Notational conventions

Terminal symbols are set in bold Courier New font

Nonterminal symbols are set in italic Arial font

Optional parts are marked by a dashed underline

- Logic meta-variables can be contained in the expansion of
  - *pointcut*, *introBody*, *adviceBody*
  - *params*, *ex*, *interface*

Uniform Genericity

- Use of pointcuts in all aspect language elements
  - introductions, advice, declarations
- Use of meta-variables in all aspect language elements
  - pointcut, introBody, adviceBody, params, ex, interface

➔ Context-dependent aspect effects

- Meta-variables are bound in pointcuts providing values for the body of introductions / advice / declarations

Comparison

Comparison

Uniformly Generic Aspects	Framed Aspects
Language integrated genericity	External macro system
One programming paradigm	Two programming paradigms
Parameters enumerated in concrete aspects	Parameters enumerated in external representation
Parameters specified by program analysis (pointcut evaluation)	No analysis capability
Context-dependent variation	No context notion
Open ended variations	Fixed enumeration of variations
→ more expressive	→ externally configurable

Approaches could be combined…

Other Generic Aspect Languages

- Jats (Java Transformation System)
  - Concrete syntax
  - Pointcuts are source patterns
  - Only one source pattern for each transformation
- Carma (Smalltalk)
  - Only generic advice
  - Based on SOUL (Logic Meta-Programming Framework for Smalltalk)
- Sally (Java)
  - First version only featured parametric (generic) introductions
  - Now very similar to LogicAJ

Not supported anymore

Other Generic Aspect Languages

- Meta-AspectJ
  - generates AspectJ aspects via meta programming
  - mixes concrete syntax with abstract syntax tree manipulations

- Detailed comparison and more related work
  - G. Kniesel, T. Rho: “A Definition, Overview and Taxonomy of Generic Aspect Languages”, *L'Objet, vol. 11, 3*, Hermes Science, London, 2006.
  - http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/knieselRho-lObjet2006.pdf

LogicAJ References

- Paper
  - Tobias Rho, Günter Kniesel: "Uniform Genericity for Aspect Languages", *Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn*
  - http://roots.iai.uni-bonn.de/research/logicaj/downloads/papers/RhoKniesel-IAI-TR-2004-4.pdf
- Website
  - http://roots.iai.uni-bonn.de/research/logicaj/