Fundamentals of Concern Manipulation

Harold Ossher
IBM T. J. Watson Research Center

The CME Team
(IBM Hursley Park and IBM Watson)
William Chung, Andrew Clement, Matthew Chapman,
William Harrison, Helen Hawkins, Sian January, Vincent Kruskal, Harold Ossher, Stanley Sutton, Peri Tarr, Frank Tip

This research was supported in part by the Defense Advanced Research Projects Agency under grant NBCHC020056
Copyright © IBM Corporation, 2002, 2007
Contents

- Concerns
- Concern Manipulation
- Core concepts
  - Artifact representation
  - Concerns
  - Query
  - Composition
- Conclusion
Concern

- Area of interest in a body of software
- \((intension, extension)\) pair
  - intension specifies meaning
    - query, predicate
    - extension lists applicable software elements
  - extension = intension(body of software)

- Diverse
Requirements have widespread and diffuse representation in implementation.
A Requirement is a Concern

Requirements Elements

Design Elements (Components, Strategies, Patterns)

Implementation Elements (Objects, Methods)
An Object (Data) Implementation Concern

Requirements Elements

Design Elements (Components, Strategies, Patterns)

Implementation Elements (Objects, Methods)
A Component Design Concern

Requirements Elements

Design Elements (Components, Strategies, Patterns)

Implementation Elements (Objects, Methods)

Copyright © IBM Corporation, 2002, 2007
All Design Artifacts

Requirements Elements

Design Elements (Components, Strategies, Patterns)

Implementation Elements (Objects, Methods)
A Design Aspect

Requirements Elements

Design Elements (Components, Strategies, Patterns)

Implementation Elements (Objects, Methods)
Concern Manipulation

Create or Identify Encapsulate Extract Compose

Using concerns in any and all ways that are useful during development
Implications

Using concerns in any and all ways that are useful during development

**Tools**
- Diverse, but uniform experience
- Specific artifacts (perhaps)
- Specific paradigms, symmetric or asymmetric

**Core Concepts**
- Uniform, shared
- Artifact type neutral
- Paradigm neutral, symmetric

Copyright © IBM Corporation, 2002, 2007
Uniform User Experience

• Uniform user experience across tools, e.g.,
  – First-class, ubiquitous concerns
  – Central concern model
  – Same queries for exploring, mining, defining concerns, pointcuts, composition, …

⇒ Uniform set of core concepts
  – Supported by shared underlying components

Copyright © IBM Corporation, 2002, 2007
Artifact Type Neutrality

• In realistic systems, concerns include elements from artifacts of different kinds:
  – Requirements
  – Design
  – Code
  – Documentation, Help files
  – Data files (XML, properties, icons, …)
  – Deployment artifacts (JAR, WAR, ZIP, …)
  – …

• Large, open set

⇒ Core concepts should be artifact type neutral
Paradigm Neutrality

- Many AOSD paradigms
  - Symmetric and asymmetric
  - Static and dynamic join points
  - Member level and code level join points
  - Various query paradigms (e.g., patterns, logic, …)
  - …

⇒ Core concepts should be paradigm-neutral
  - General enough to support multiple paradigms
  - Some may be for the tool implementer, rather than user
    • Paradigm-specific concepts are surfaced to the user by tools
Symmetry

- Is there a distinguished base?
  - E.g., aspect applied to base, or peer concerns being composed
  - Are elements being composed of same kind?
  - Paradigms differ in their choice
  - Important scenarios for both
    - Impact of scenario-model mismatch is significant

⇒ One model that can handle both:
  - Symmetric underlying model
  - Asymmetric façade
Symmetric Model, Asymmetric Facade

• Symmetric model supports general concerns ➔ concern composition
  – Works for concerns that are peers or in base-aspect (or base-extension) relationship
  – Paradigm-neutral

• Asymmetric Façade critical for convenience:
  – Paradigm-specific, supported by tools
  – Translate to symmetric model
Artifact Model

Core concepts for representing different kinds of artifacts

• Entities
  – Modifiers, Classifiers
  – Attributes
• Relationships

• Container spaces
  – Containers
    • Elements
• Type spaces
  – Types
    • Fields
    • Methods

Spaces are *declaratively complete* (contain definitions of names used)
E.g., Java classpath, collection of UML model files

Copyright © IBM Corporation, 2002, 2007
Space Example

- **Base** defines the representation of employees
- **Report** implements a reporting feature
Intertype Declaration

class Emp {
    int id() { … }  
    String name() { … }  
    …  
}

aspect Report {
    void Emp.print(OStream o) {
        o.println(id() + "": " + name());
    }
}
Declarative Completeness: Abstract

space Base

class Emp {
    int id() { … }
    String name() { … }
    …
}

library space

Symmetric

class String { … }

space Report

abstract class Emp {
    void print(OutputStream o) {
        o.println(id() + ":: " + name());
    }
    abstract int id();
    abstract String name();
}

Symmetric

Abstract

class Emp {
    int id() { … }
    String name() { … }
    …
}
Declarative Completeness: \textit{Requires}

\hspace{Base}
\begin{verbatim}

class Emp {
    int id() { … }
    String name() { … }
    …
}

\end{verbatim}

\hspace{Report}
\begin{verbatim}

requires int Emp.id();
    String Emp.name();

class Emp {
    void print(OStream o) {
        o.println(id() + ":: " + name());
    }
}

\end{verbatim}

\hspace{library space}
\begin{verbatim}

class String { … }
    …
\end{verbatim}

\textbf{Symmetric}
Included Spaces?

space Base

class Emp {
    int id() { … }
    String name() { … }
    …
}

space Report

class Emp {
    void print(OStream o) {
        o.println(id() + "::" + name());
    }
}

library space

class String { … }

Symmetric/
Asymmetric
Synthesis
Language Binding

• To be applied, these concepts must be bound to actual artifact types:
  – E.g., file system:
    • Container space $\rightarrow$ Root Directory
    • Container $\rightarrow$ Directory
    • Element $\rightarrow$ File
  – E.g., Java
    • Type space $\rightarrow$ Class path
    • Type $\rightarrow$ Class, Interface, …

• In practice: artifact-type-specific plug-ins
• Use patterns to define material inside element bodies, treating the matching material as extractable elements

class C {
    int x;
    void foo() {
        ... 
        x = 3;
        ...
        x = y+7;
    }
}

methoid setX:
kind = “set”
field = “x”

class C {
    int x;
    void foo() {
        ...
        setX(3);
        ...
        setX(y+7);
    }
    void setX(int x) {
        this.x = x;
    }
}
• Allow uniform handling of code-level join points
  – Methoid occurrences are elements for searching, composition, …
• Open-ended characterizations (mapping-specific)
  – E.g., useful language constructs:
    • get/set of specific instance variables
    • method calls, entries and exits
    • synchronization block entries and exits
    • throws and catches of specific exception types
    • downcasting and instanceof
  – Can specify arguments, set to local state
    • Perhaps specially-constructed (e.g., thisJoinPoint)
• Various inlining options and, perhaps, restrictions
  (mapping-specific)
Correspondence

• Elements to be composed together

Base

class Emp {
    int id() { … }
    String name() { … }
    …
}

Log

aspect Log {
    pointcut ops(): execution (* Emp.*(..)):
    before (): ops() { logEntry(); }
    void logEntry() { … }
}

{ (Emp.id, Log.logEntry), (Emp.name, Log.logEntry) }
Correspondence

- Elements to be composed together

```java
space Base

class Emp {
    int id() { … } 
    String name() { … }
    …
}

merge (Emp.*, Log.logEntry)

Correspondence Query

{ (Emp.id, Log.logEntry), (Emp.name, Log.logEntry) }
```
Concerns

• Intension (query) and extension (set of elements)
• First-class
  – Explicitly modeled, used for exploration, composition, …
  – Relationships among concerns
  – Composition relationships
• Heterogeneous
• Written explicitly as modules, or mined
## Concern versus Space

<table>
<thead>
<tr>
<th>Concern</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any elements</td>
<td>Containers/Types</td>
</tr>
<tr>
<td>No name usage restrictions</td>
<td>Declaratively complete</td>
</tr>
</tbody>
</table>

Extraction
Query

- **Diverse query languages**
  - Each usable wherever desired

- **Core concepts**
  - Selection of:
    - Elements, based on names, modifiers, classifiers, attributes, containment
    - Methods, based on their patterns
    - Relationships, based on their names and characteristics of their end points
    - Correspondences
  - Navigation via relationships
    - Including transitive closure
  - Predicates and set operations
  - *Variables and unification*
    - E.g., (class p1.<C>, class p2.<C>)
Simple Composition Example 1

basic

```java
class Sys
    int interval;
    void init() {...};

class RoomSensor
    void report() {...};
    void update(int) {...};

class AtticSensor
    void report() {...};
    void update(float) {...};

... more sensors ...
```

alarm

```java
aspect Alarm
    after execution(
        **.update(int)): {...};
    after execution(
        **.update(float)): {...});

composed result 1

```java
class RoomSensor
    Alarm a = ...;
    void u_b(int) { /* basic*/
        void update(int i) {
            u_b(i); a.update(i); }
    ...
```
**Simple Composition Example 2**

```plaintext
basic

```class Sys
   int interval;
   void init() { ...};

```class RoomSensor
   void report() { ...};
   void update(int) { ...};

```class AtticSensor
   void report() { ...};
   void update(float) { ...};

... more sensors ...

```alarm

```class Alarm
   void update(int);
   void update(float);

```composed result 2

```class RoomSensor
   void u_b(int) { /* basic*/}
   void u_a(int) { /* alarm*/}
   void update(int i) {
      u_b(i); u_a(i); }

... }```

Copyright © IBM Corporation, 2002, 2007
Simple Composition Example 3

```java
class Sys
    int interval;
    void init() {...};

class RoomSensor
    void report() {...};
    void update(int) {...};

class AtticSensor
    void report() {...};
    void update(float) {...};

... more sensors ...
```

merge ...

```java
class Alarm
    void update(int);
    void update(float);

class RoomSensor
    Alarm a = ...;
    void u_b(int) { /* basic*/}
    void update(int i) {
        u_b(i); a.update(i); }

... }
```
Dynamic Join Points

• Dynamic join points are typically handled by:
  – Generation of dynamic residues
  – Static composition (at join point shadows)
Levels of Composition Specification

Tool Level

Component Level

Assembly Level

merge basic, alarm as C
merge order(1, 2) facet:
  space basic, alarm as C
  encapsulating(member)
  exposed
  exclusively precedence(1)

<type name="Sys" attributes="public"/>
<method within="C:Sys" name="init" types="()">
  <from within="basic:Sys" name="init" types="()"/>
</method>
<field within="C:Sys" name="interval" type="int">
  <from within="basic:Sys" name="interval" type="int"/>
</field>

Copyright © IBM Corporation, 2002, 2007
Weaving Directives

- What elements are to be joined?
  - Correspondence

- How are they to be joined
  - Selection
  - Ordering
  - Structure

- Making assumptions explicit
  - Encapsulation, Opacity

- Resolving multiple weaving directives
  - Exclusivity, Precedence

merge order(1, 2) facet:
  space basic, alarm as C
  encapsulating(member)
  exposed
  exclusively precedence(1)
Identifying Correspondences

• Explicit: queries
  – (class basic:*Sensor, alarm:Alarm)
    
    \{(RoomSensor, Alarm), (AtticSensor, Alarm)\}
  
    \{(RoomSensor.update(<type>), alarm:Alarm.update(<type>))\}
  
    \{(RoomSensor.update(int), Alarm.update(int)),
          (AtticSensor.update(float), Alarm.update(float))\}

• Implicit (depending on encapsulation)
  – Like-named types within corresponding spaces
  – Like-named members within corresponding types
Selection

- *Which* inputs in the correspondence should participate in the result:
  - merge
  - override
  - overridemember
  - aroundmethod
  - any
  - unique
  - … (this is an open-ended list)
Ordering

• For override/around: which input dominates
• For merge of methods: order of execution
  – Generalized as method combination graphs

(method basic:*Sensor.update(<type>),
   after:: alarm:Alarm.update(<type>))
Method Combination Graphs

- Nodes call methods or exit
- Various choices for arguments
  - Incoming arguments, return values
  - Target and its instance variables (e.g., aspect or role table)
  - Static variables, special "meta" variables
- Various conditions on edges
  - Normal return versus exception
  - Some value checks on variable values (allows multiple dispatch)
- Call auxiliary methods for complex processing
- Non-determinism, supporting composability

![Diagram]

Copyright © IBM Corporation, 2002, 2007
Structure

• *How* inputs participate in the result
  – How do the lifetimes and identities of the participants relate? E.g.:
    • Single result type or collaborating group (e.g., object & aspect)?
    • Do references to the input map to the output or not?
  – How is *this* treated? To what does it refer?
  – What happens to *static*?
  – What are the linkage conventions?
• facet, copy, aspect, … (another open point)
• FOAL ‘02 paper on “member-group relationships”
Opacity

• Is the type hierarchy structure assumed to be known and taken into account?

\[ a + b \quad \Rightarrow \quad ab \quad \text{the space is opaque} \]

\[ a \quad \Rightarrow \quad b \quad \Rightarrow \quad a + b \quad \Rightarrow \quad ? \quad \text{the space is exposed} \]
Levels of Composition Specification

Tool Level

Component Level
merge basic, alarm as C
merge order(1, 2) facet:
space basic, alarm as C
encapsulating(member)
exposed
exclusively precedence(1)

Assembly Level

Paradigm-specific
Paradigm-neutral

<type name="Sys" attributes="public"/>
<method within="C:Sys" name="init" types="()">
  <from within="basic:Sys" name="init" types="()"/>
</method>
<field within="C:Sys" name="interval" type="int">
  <from within="basic:Sys" name="interval" type="int"/>
</field>

...
public class Driver {

public void f(String arg) {
    System.out.println("body of f()");
}

public void g(String arg) {
    System.out.println("body of g()");
}
}

public class Trace {

public void before(Object arg, String mName) {
    System.out.println(">>> before ");
}

public void after(Object arg, String mName) {
    System.out.println(">>> after ");
}
}

public class Driver {

public void original_f(String arg) {
    System.out.println("body of f()");
}

public void g_original_g(String arg) {
    ...}

public void trace_before(Object arg, String mName) {
    System.out.println(">>> before ");
}

public void trace_after(...) {
    ...
}

public void f(String arg) {
    trace_before(arg, "Driver.f");
    original_f(arg);
    trace_after(arg, "Driver.f");
}

public void g(String arg) {
    ...
}
Conclusion

- **Core concepts for concern manipulation**
  - Artifacts, concerns, queries and composition
  - Artifact-type neutral
  - Paradigm neutral

- **Wide open research area**
  - Validation and improvement
    - Mapping (and implementing) different paradigms
    - Mapping (and implementing) more artifact types
  - Asymmetry, included spaces, concern/space relationship
  - Extraction
  - New issues, e.g., versioned artifacts, dynamic AOSD
  - ...

Copyright © IBM Corporation, 2002, 2007
Thank you!