

Separation of Concerns Techniques

Morgan Deters
mdeters@cs.wustl.edu



www.cs.wustl.edu/~mdeters/

Spring 2002 Programming Languages Seminar

23 January 2002

SoC: How to get there from here

- **Post-Object Programming (POP)**
 - Generic programming, generative programming, reflection/metaobject programming, ...
- **Need for modularizing *crosscutting concerns***
- **Techniques used to achieve “post-object” separation of concerns**
 - **Aspect-Oriented Programming**
 - **Composition Filters**
 - **Adaptive Methods**
 - **Metaobject Programming**
 - **Subject-Oriented Programming**
 - **Hyperspaces**

MOP versus ASoC

- **Metaobject Protocol**
 - power
 - reflection
 - adaptability
 - specialization
- **“Advanced” Separation of Concerns**
 - modularity of (otherwise) crosscutting concerns
 - obliviousness
 - componentization
 - abstract patterns

Crosscutting concerns

- “The crucial choice is, of course, what aspects to study ‘in isolation,’ how to disentangle the original amorphous knot of obligations, constraints and goals into a set of ‘concerns’ that admit a reasonably effective separation.” - E.W. Dijkstra
- Object decompositions can’t totally encapsulate synchronization, real-time, scheduling, transaction semantics, caching/prefetching strategies, memory management concerns – these are *systemic*
- Similarly, there are *functional* crosscutting concerns
 - business rules, logging (maybe), other features

Characteristics of AOP

- AOP systems “provide quantification and obliviousness” [Filman]
- Quantification - particular statements affect various different places in the code
- Obliviousness (or “implicit invocation”) - quantifications affect places in the code that are not specifically marked – they are *oblivious* to the quantifications they receive.

AOP at a Glance

- *aspect* - loosely, a crosscutting concern (or its realization)
- *join point* - a well-defined point in a program
 - could be *data* or *execution*, *static* or *dynamic*
- *advice* - quantifications to apply
- *weaving* - the process by which aspects exert influence
- *AOP supplements OOP*, does not supplant it!

AspectJ

- *execution join points*
 - method and constructor executions and calls
 - class and instance variable access and mutation
 - control flow
- *aspect* - a crosscutting *type* that encapsulates state and behavior about a particular crosscutting *concern*
- *advice* - the quantifications to apply
- *pointcuts* - join points on which to apply quantifications
- *lexical introduction* - static extension of behavior

Example: A logging aspect

```
public abstract aspect Logger {
    abstract pointcut loggableEvents();    /* events to log */

    public PrintStream log = System.err;

    /* don't log the Logger! */
    pointcut events(): loggableEvents() && !within(Logger+);

    before(): events() {
        log.println("BEFORE " + thisJoinPoint);
    }
    after() returning(): events() {
        log.println("SUCCESS " + thisJoinPoint);
    }
    after() throwing(): events() {
        log.println("FAILURE " + thisJoinPoint);
    }
}
```

Extending the Logging Aspect

```
/* abstract Logger */
public abstract aspect Logger {
    abstract pointcut loggableEvents();
    /* etc */
}

/* one possible specialization of Logger */
public aspect Flogger extends Logger {
    pointcut loggableEvents(): call(* *.*(..));
}

/* another specialization of Logger */
public aspect DOCLogger extends Logger {
    pointcut loggableEvents(): execution(* edu.wustl.doc...*(..));
}
```

More AOP

- Tracing/logging is a simple example, what else?
 - synchronization/locking strategies
 - modularization of features
 - security
 - customization
 - debugging support
 - real-time correctness
 - power constraints
- Problems using AOP
 - Composition of dependent aspects (advising other aspects)
 - Not always the right paradigm!

Composition Filters - Sina/st

- Filters enhance objects in *modular* and *orthogonal* ways by manipulating messages sent to and from the object
 - *modular* - independent of implementation language of target object
 - *orthogonal* - independent of each other
 - A CF class consists of filters and internal classes (which may be CF classes)
- Filters can be used to provide delegation, protection, realtime scheduling
- Filter types are *dispatch*, *error*, *wait*, *meta*, *realtime*
- Filters can achieve multiple views, multiple inheritance

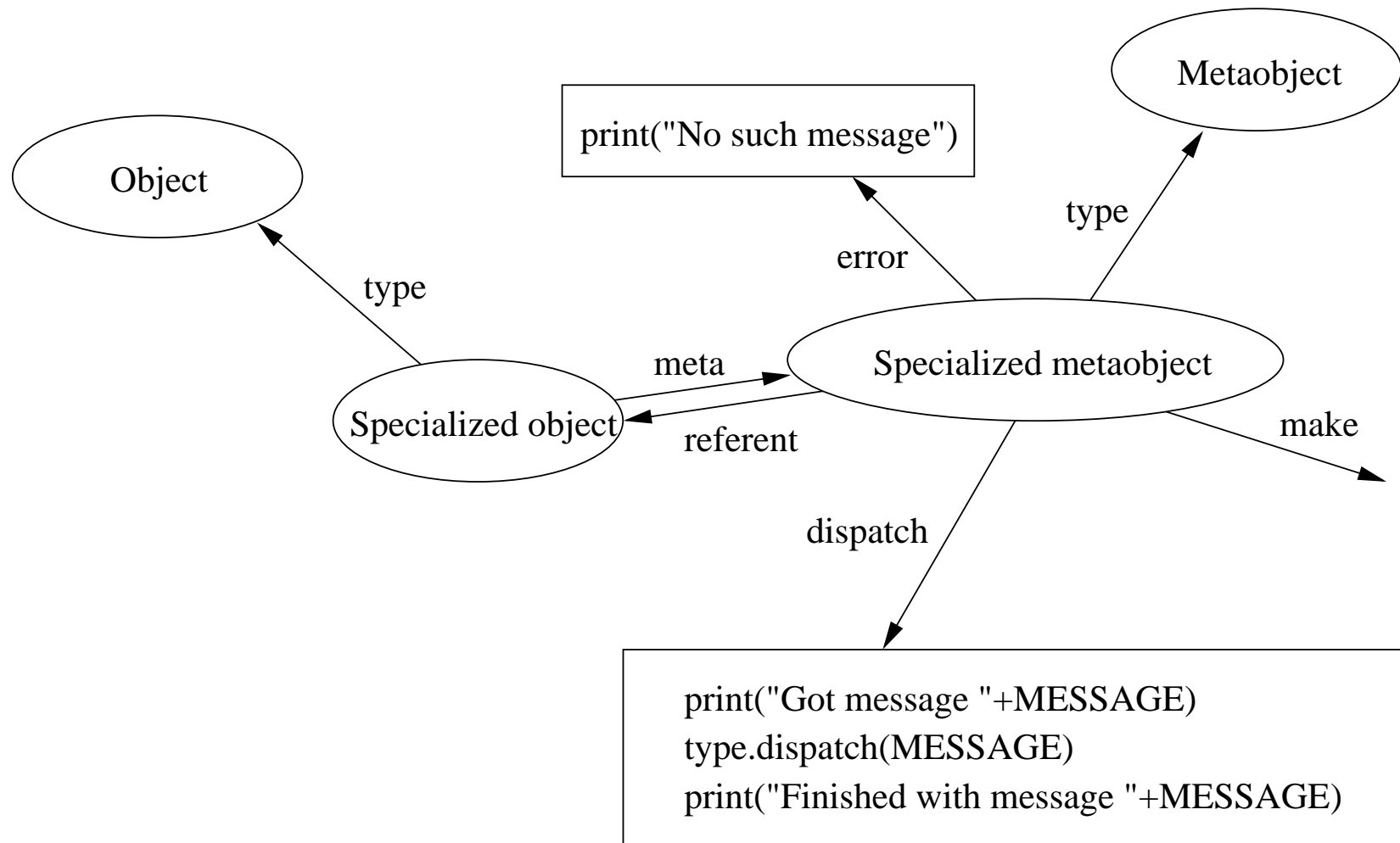
Adaptive Methods - Demeter/Java

- Objects maintained in a UML-like class graph
- Adaptive methods technique consists of:
 - class graph
 - adaptive methods (aspects)
 - traversal strategies (collection of join points)
 - adaptive visitors (advice)
- Join points are not points during execution but edges in class graph

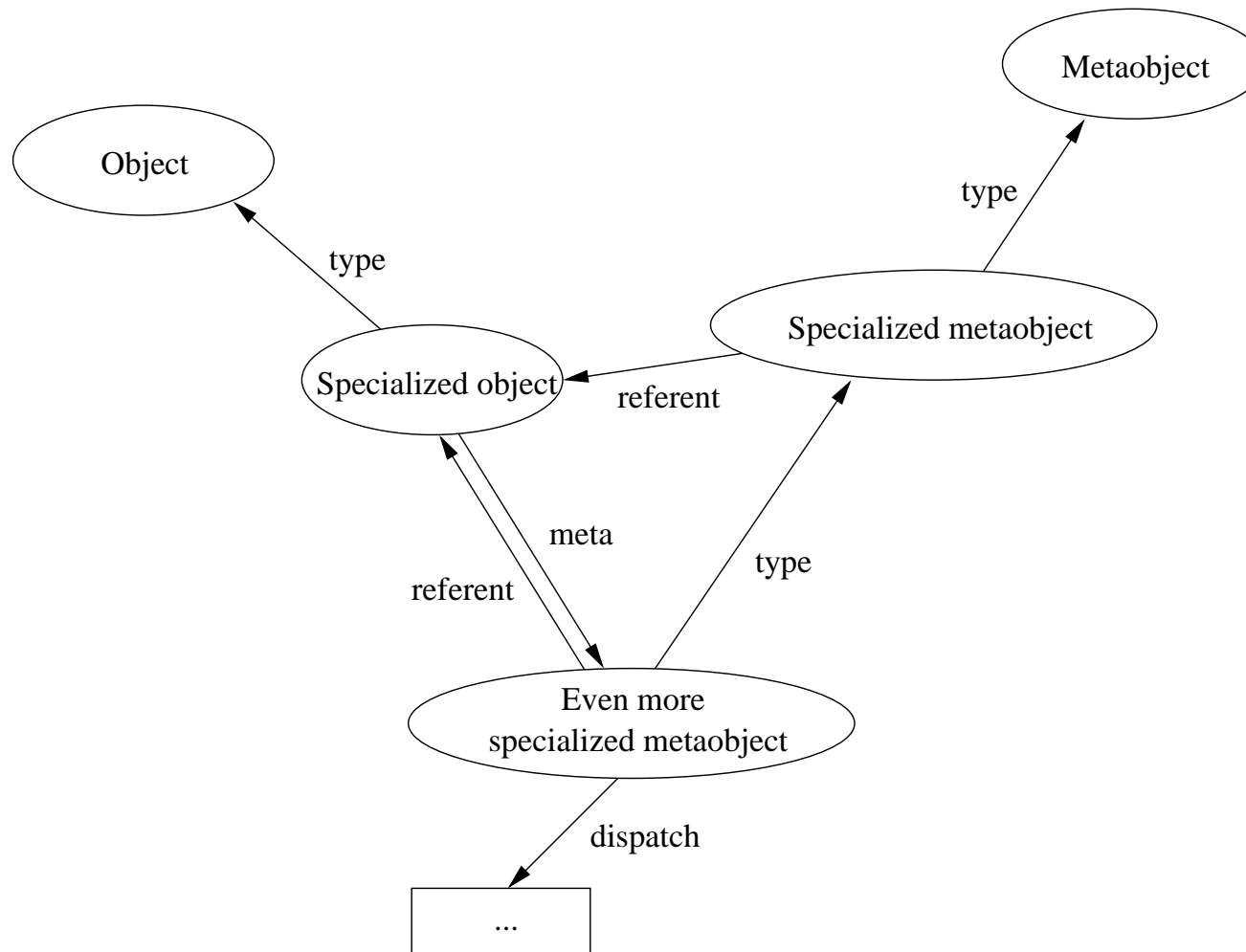
Subject-Oriented Programming

- View objects differently based on viewpoint
 - Interobject relationships have subject bias
- IBM's SOP work → hyperspaces research
- More on this next week!

Structure of MOP - review



Composition (aspect weaving?) using MOP



Discussion



Washington
University in St. Louis

Spring 2002 Doctoral Seminar on
Programming Languages

23 January 2002