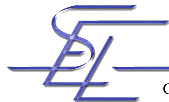
COMP-667 Software Fault Tolerance

Overview of AspectOPTIMA

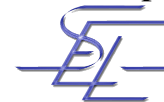
Jörg Kienzle
School of Computer Science
McGill University, Montreal, QC, Canada

With Contributions From:
Samuel Gélineau, Ekwa Duala-Ekoko,
Güven Bölükbaşi, Barbara Gallina



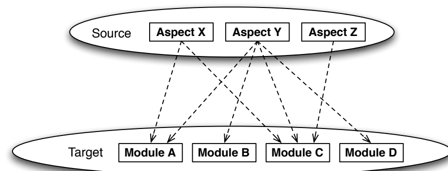
Talk Outline

- Background on AOP
- My view of the "Essence" of Aspect-Orientation
  - Weaving, Scattering, Tangling, Crosscutting
- Aspects and Reuse
- AspectOPTIMA
  - Aspects for Objects, Threads and Contexts
  - Example Configurations
- Conclusion
- Exception Handling Extension



Aspect-Orientation

- Aspect-oriented software development (AOSD) techniques aim to provide systematic means for the identification, separation, representation and composition of *crosscutting* concerns

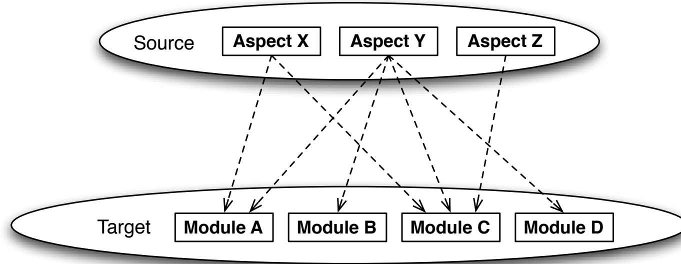


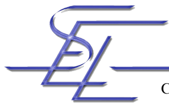
Aspect-Oriented Programming

- Modularize crosscutting concerns at the programming language level
- Decompose problem into aspects, encapsulating different concerns of the application [K+97]
- *Weave* aspects together for final product
- Weaving happens at so-called *joinpoints*
- Benefits: Simpler structure, improve readability, customizability and reuse

Weaving

- Mapping from a source representation to a target representation

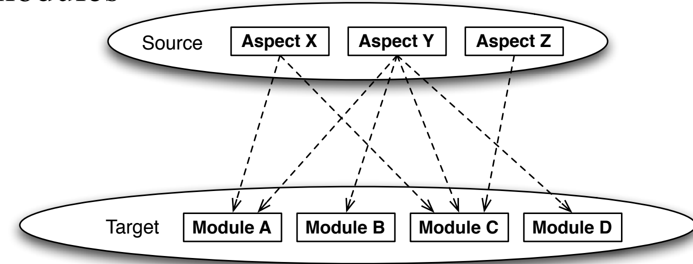


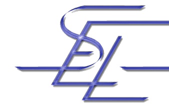


McGill

Scattering

- A source module is scattered in a target representation if part of it ends up in many target modules

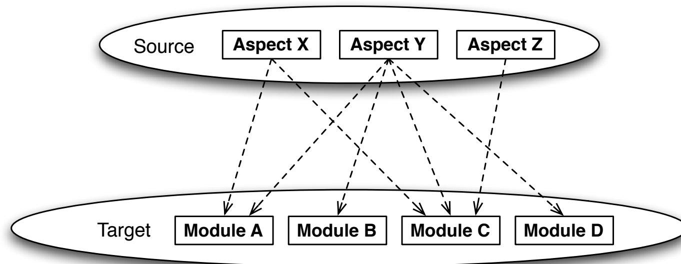




McGill

Tangling

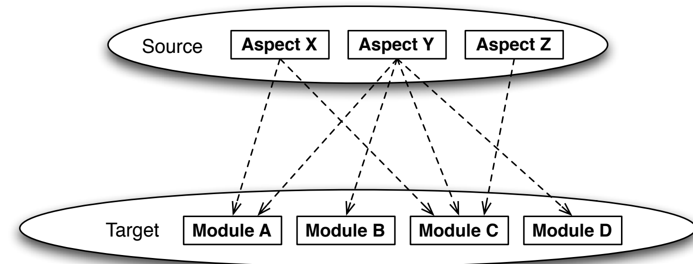
- A target module is tangled if it is composed of parts of several source modules



McGill

Crosscutting

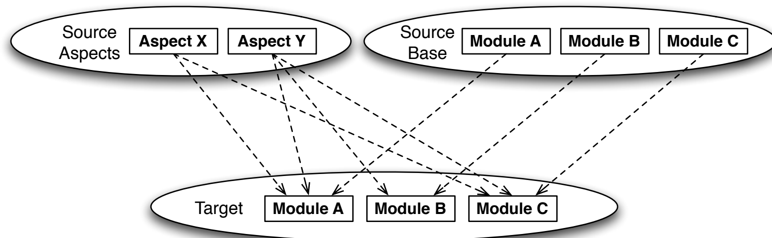
- X crosscuts Y iff X is scattered in the target representation, and there exists a module in the target within which X and Y are tangled

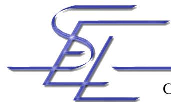


McGill

Asymmetric AO

- Well-identified base elements do not (and are not allowed to) crosscut

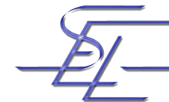




McGill

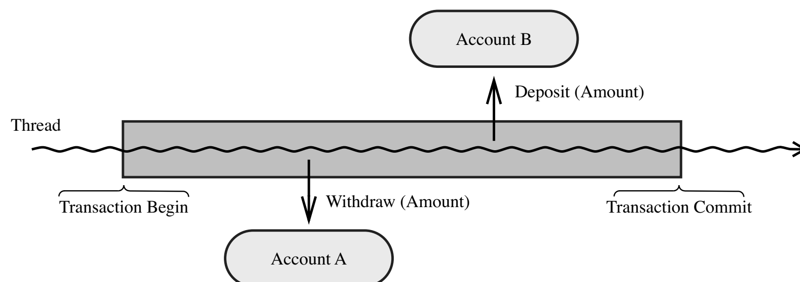
Aspect Case Study: Transactions

- A transaction groups together a set of operations on data objects, guaranteeing the ACID properties
  - Atomicity
  - Consistency
  - Isolation
  - Durability



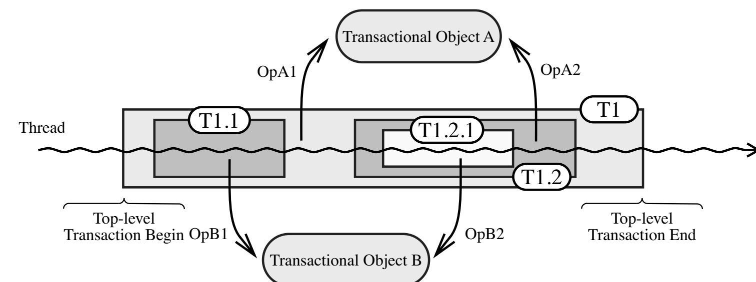
McGill

Flat Transactions



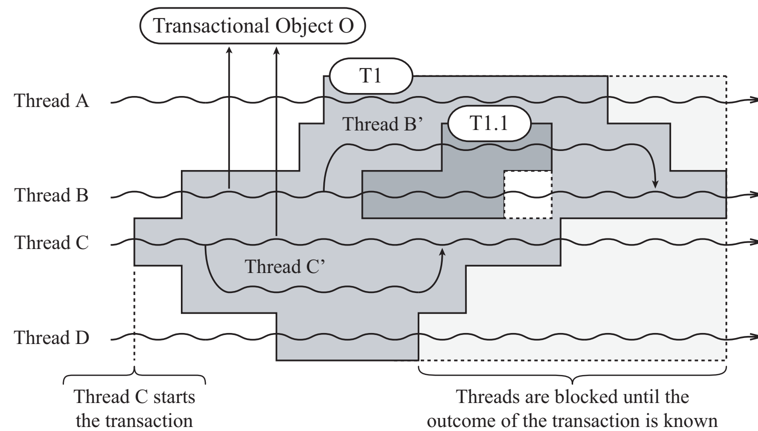
McGill

Nested Transactions



McGill

Open Multithreaded Transactions

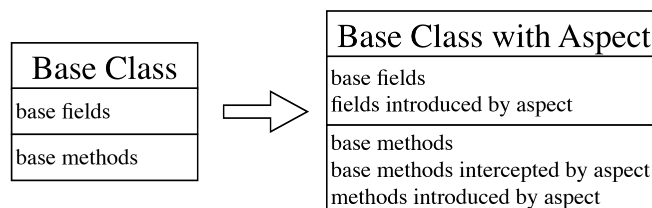


AspectOPTIMA

- Observations
  - Concurrency control and recovery are separate concerns at a higher level of abstraction
    - At the implementation level, the two concerns are tightly coupled
  - Most transaction models are related, i.e. they share common concepts
- Challenge
  - Is it possible to define many individually reusable aspects that, when put together in different ways, can implement various transaction models, concurrency control and recovery strategies?

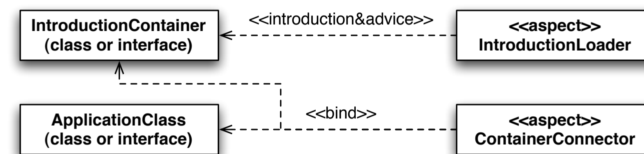
AspectJ Design

- In our AspectJ implementation of AspectOPTIMA, an aspect encapsulates additional structure and behavior applicable to base classes



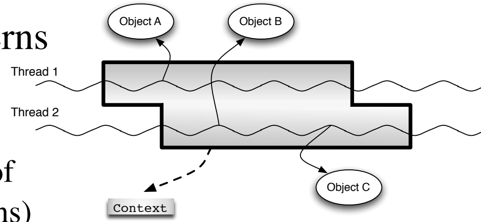
Reusable Bindings in AspectJ

- Abstract Introduction Idiom
  - Each aspect is applied to a dummy interface
  - Bindings are established by making an application class implement the dummy interface
    - Binding can be specified using an aspect as well!

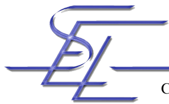


Design of AspectOPTIMA

- 3 high-level concerns
  - *Objects*
  - *Threads*
  - *Contexts* (or scopes of computations)

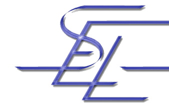


- We identified 12 aspects for objects, 3 for threads, and 13 for contexts
  - Each aspect has well-defined functionality and is individually reusable
  - Subtle dependencies and conflicts between aspects



AccessClassified in AspectJ

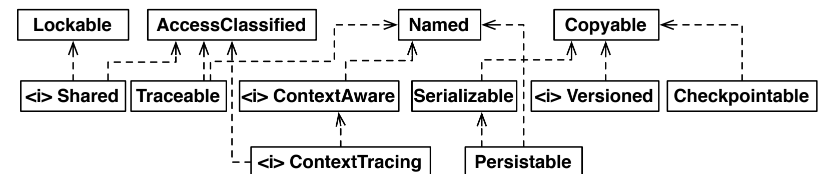
```
package library.aspects.object;
import java.lang.reflect.Method;
import library.annotations.*;
import library.interfaces.AccessClassified;
import library.util.AccessTypes;

public aspect AccessClassifiedAspect {
  public String AccessClassified.getAccessTypeOfMethod (String methodName) {
    String accessType = AccessTypes.WRITE;
    for (Method method : this.getClass().getMethods()) {
      if ((methodName.trim()).equalsIgnoreCase(method.getName())) {
        if(method.isAnnotationPresent(ReadAccess.class))
          accessType = AccessTypes.READ;
        else if(method.isAnnotationPresent(UpdateAccess.class))
          accessType = AccessTypes.UPDATE;
        break;
      // If there is no annotation, assume the worst case
      }
    }
    return accessType;
} }
```



AccessClassified Bank Account

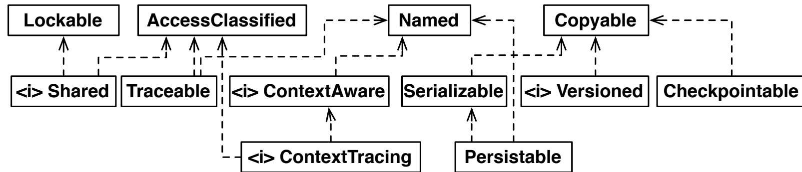
```
import library.annotations.*;
import library.interfaces.AccessClassified;

public class Account implements AccessClassified {
    public Account(int startingBalance) {
        balance = startingBalance;
    }

    @ReadAccess
    public int getBalance()
    {
        return balance;
    }

    @WriteAccess
    public void setBalance(int newBalance)
    {
        balance = newBalance;
    }

}
```

Per Object Aspects

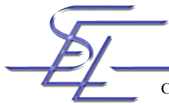


- *Lockable*: Creates lock types, gets and releases locks
- *AccessClassified*: Provides access kind for each method (*read*, *write*, *update*)
- *Named*: Associates a name (string) with each application object instance

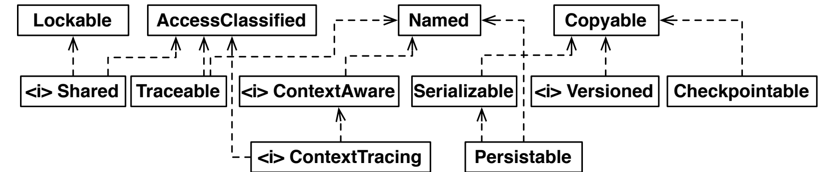
Per Object Aspects



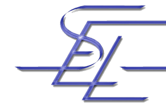
- *Copyable*: Provides cloning and state replacement capabilities
- *Shared*: Enforces multiple reader / single writer
- *Traceable*: Provides operation invocation information



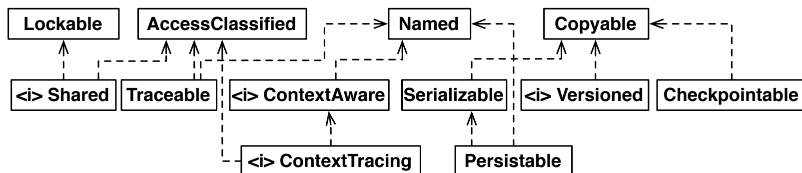
Per Object Aspects



- *ContextAware*: Informs context whenever an operation is invoked
- *Serializable*: Provides streaming capabilities
- *Versioned*: Creates views (separate instances of the *same* application object), associable to threads



Per Object Aspects



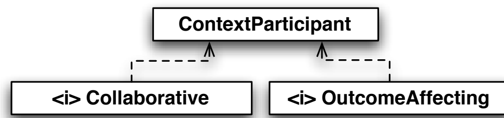
- *Checkpointable*: Establishes, restores and discards checkpoints
- *ContextTracking*: Remembers contexts that access the object
- *Persistable*: Saves and loads state from stable storage

Specifying Dependencies in AspectJ

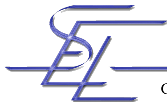
- Traceable objects have to be AccessClassified and Named as well

```
package library.aspects.object;
import library.interfaces.Traceable;
import library.interfaces.Named;
import library.interfaces.AccessClassified;
import library.util.ObjectTrace;

public aspect TraceableAspect {
   declare parents: Traceable implements Named, AccessClassified;

   public ObjectTrace Traceable.createMyTrace(String methodName) {
      String accessType = getAccessTypeOfMethod(methodName);
      ObjectTrace trace = new ObjectTrace(this,accessType);
      return trace;
   }
}
```

Per Thread Aspects

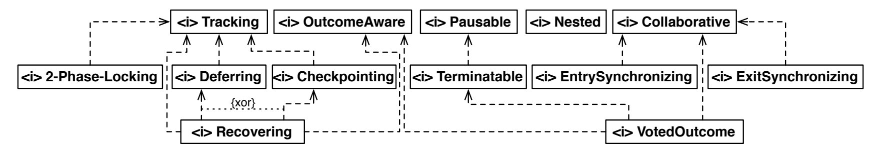


- *ContextParticipant*: Provides context creation and destruction functionality
- *Collaborative*: Provides joining functionality and control on number of participants
- *OutcomeAffecting*: Provides opinion on outcome of the context



McGill

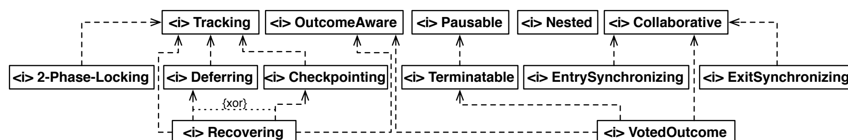
Per Context Aspects



- *Tracking*: Remembers all operation invocations made on behalf of the context
- *OutcomeAware*: Associates success/failure outcome with a context
- *Pausable*: Suspends participant work if needed

McGill

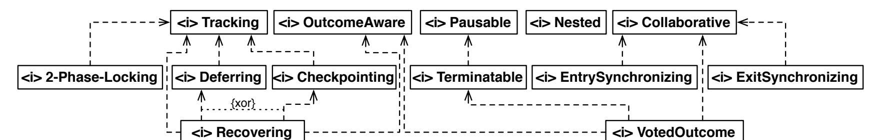
Per Context Aspects



- *Nested*: Allows contexts to be nested
- *Collaborative*: Manages many participants for a context
- *2-Phase-Locking*: Forces participants to acquire read/write/update locks when performing work, and releases all locks when context ends

McGill

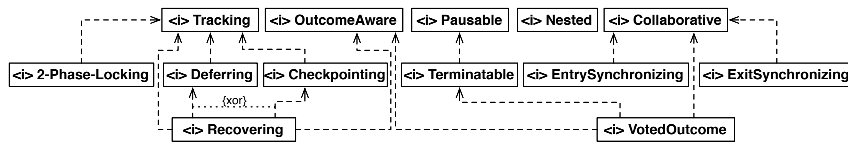
Per Context Aspects



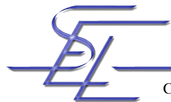
- *Deferring*: Create a context-local version of every object before modification takes place
- *Checkpointing*: Establish a checkpoint before modification takes place
- *Terminatable*: Interrupt participants and end context, if needed

McGill

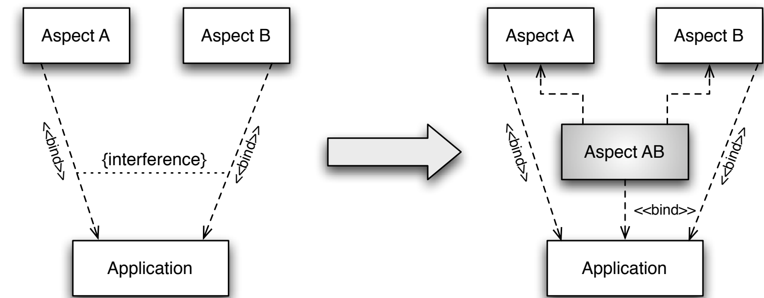
Per Context Aspects



- *EntrySynchronizing / ExitSynchronizing*: Synchronize partitipants on context entry or context exit
- *Recovering*: Undo all state changes if outcome is unsucessful
- *VotedOutcome*: Decide on context outcome by applying a voting strategy to the opinions of participants



Dealing with Aspect Conflicts



Conflict Examples

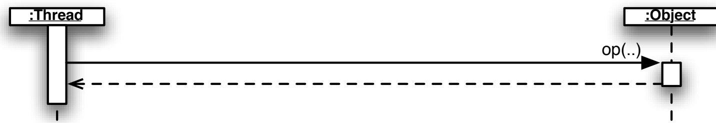
- Copyable ↔ Lockable
- Copyable ↔ Shared
- Serializable ↔ Named
- Versioned ↔ ContextAware
- Versioned ↔ Persistable
- Checkpointable ↔ Persistable
- Nested ↔ Tracking
- Nested ↔ Deferring
- Nested ↔ Checkpointing
- Nested ↔ 2-Phase-Locking
- Nested ↔ 2-Phase-Locking ↔ Recovering

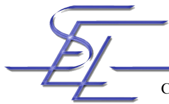
Example Configuration 1

Flat Transactions, Optimistic Concurrency Control, Deferred Update

- **Thread**: ContextParticipant, OutcomeAffecting
- **Context**: Tracking, Deferring, OutcomeAware, Recovering
- **Object**: ContextAware, AccessClassified, Named, Trackable, Copyable, Versioned, Persistable, ContextTracking

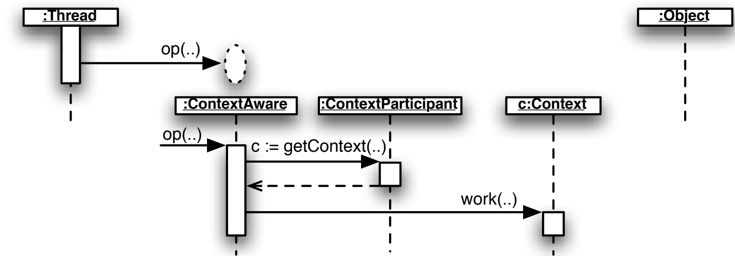
Invoking an Operation

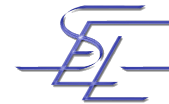




McGill

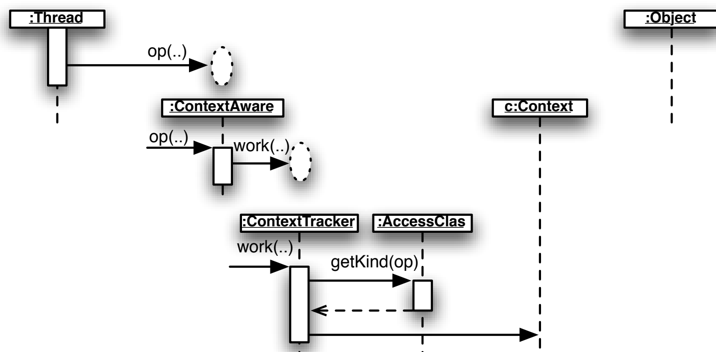
Invoking an Operation





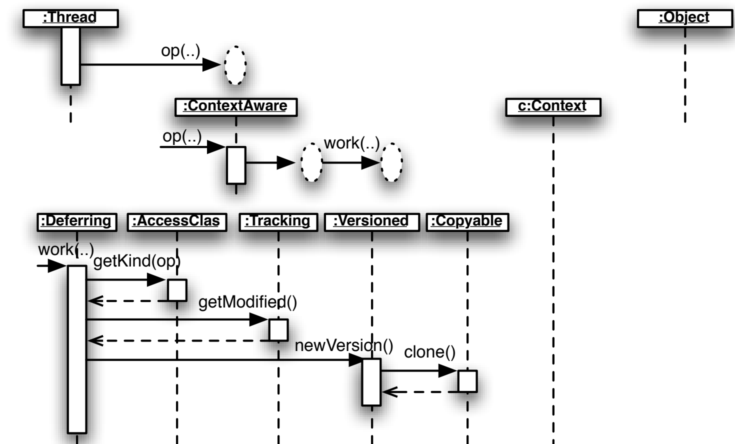
McGill

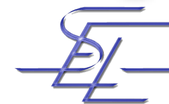
Invoking an Operation



McGill

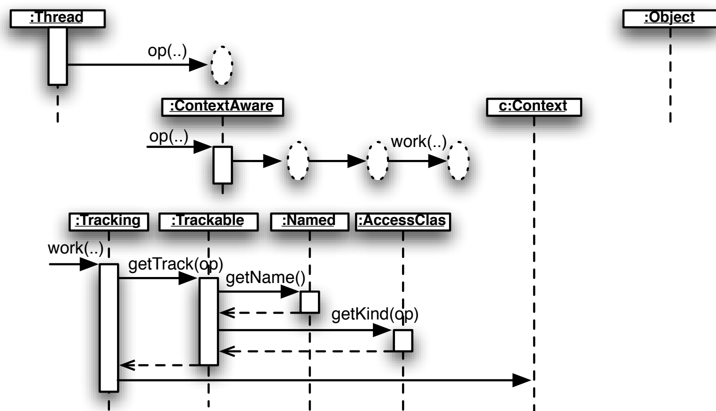
Invoking an Operation



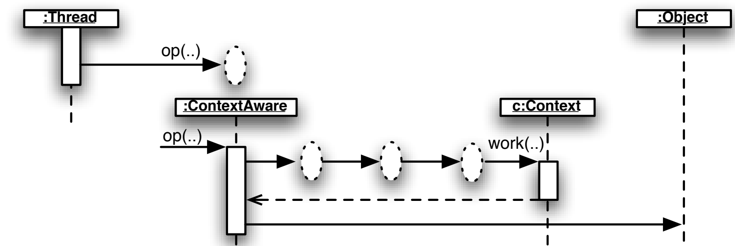


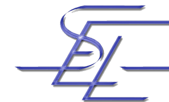
McGill

Invoking an Operation

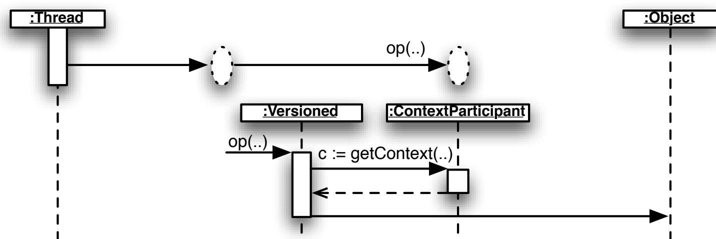


Invoking an Operation

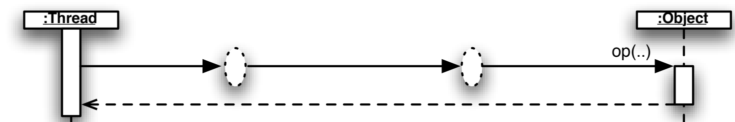




Invoking an Operation

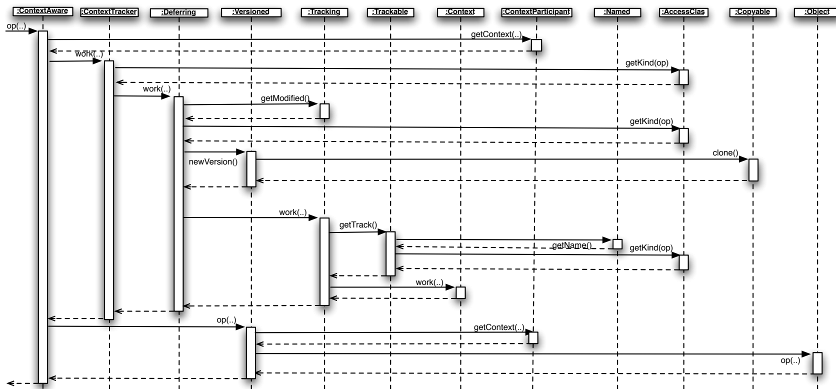


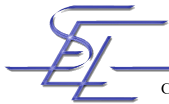
Invoking an Operation



- 5 Interceptions
  - 2 Interceptions of the actual method invocation
  - 3 Interceptions of the *work* operation of the context
- Collaboration of 11 Aspects

Operation Invocation Summary

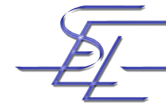




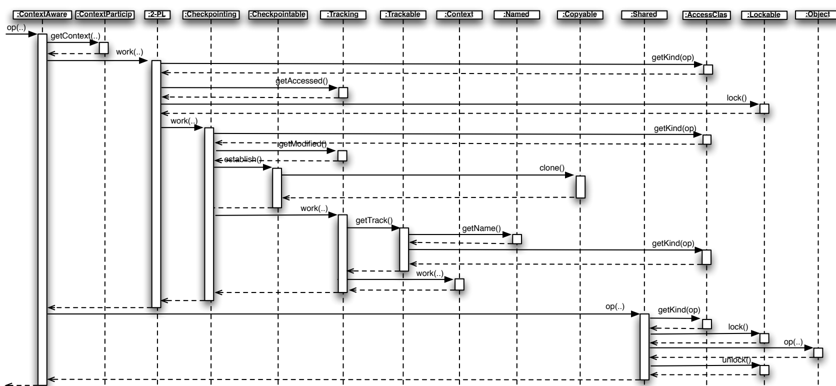
Example Configuration 2

Open Multithreaded Transactions, Pessimistic Lock-Based Concurrency Control, Inplace Update

- **Thread**: ContextParticipant, OutcomeAffecting, Collaborating
- **Context**: Tracking, 2-Phase-Locking, Checkpointing, OutcomeAware, Recovering, Nested, Collaborative, ExitSynchronizing, OutcomeVoted
- **Object**: ContextAware, AccessClassified, Named, Lockable, Trackable, Copyable, Checkpointable, Shared, Persistable

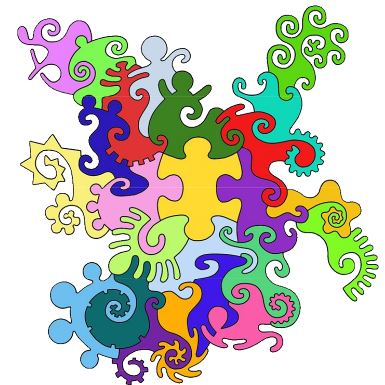


An Operation Invocation



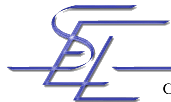
Aspect Frameworks and AO Languages

- Properties of our Design
  - Clear separation of concerns
  - High reusability
  - Complex aspect dependencies
  - Complex aspect interference
- Essential Language Features
  - Separate Aspect Binding
  - Inter-Aspect Configurability
  - Inter-Aspect Ordering
  - Per-Object (per instance) Aspects
  - Dynamic Aspects
  - Thread-Aware Aspects



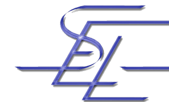
Case Study Target Audience

- Aspect-Orientation
  - AOSD Processes
  - AO Modeling Notations
  - AO Validation and Verification
  - AO Language Features
  - AO Programming Environments
- Fault Tolerance
  - Formalization of Fault Tolerance Models
  - Generation of Fault Tolerance Models



Validation by Implementation [1]

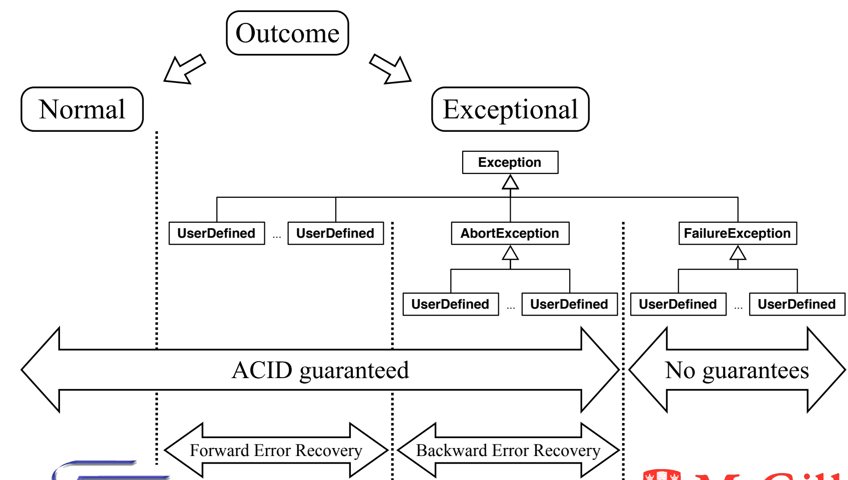
- AspectJ prototype implementation
  - Theoretical implementation in CaesarJ
- Encountered language limitations
  - Weak Aspect-To-Class Binding
  - Reflection/Superclass Execution Dilemma
  - No Explicit Inter-Aspect Configurability
  - No Per-Object Aspects
  - No Dynamic Aspects
- Work-arounds exist
- Language Improvements Suggested
- Initial performance evaluation



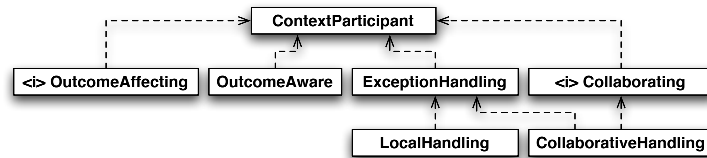
Future Work

- Define Benchmarks and Evaluate Different Compilers
- Implement AspectOPTIMA in other AO languages and compare language expressiveness
- Extend AspectOPTIMA
  - Concurrency Control and Recovery
    - Semantic concurrency control
    - Recovery based on intention lists
    - Provide weaker forms of Isolation, relaxed Atomicity
  - Transaction Models
    - Exception Handling
    - Inter-Transaction Dependencies (Look-Ahead Transactions, SAGAS)
  - Support Other Fault Tolerance Models (N-Version Programming)

Notion of Outcome



Exception Handling Participants



- *ExceptionHandling*: Capable of handling internal exceptions
- *Local Handling*: First attempts to handle internal exceptions locally
- *CollaborativeHandling*: Paricipates in collaborative handling of internal resolved exceptions
- *OutcomeAware*: Is notified of external exception

McGill

Exception Handling Objects and Context

- Context
  - BackwardRecovering and ForwardRecovering
  - ExceptionResolving
- Objects
  - Self-Checking

Exact functionality still to be determined

McGill

AspectOPTIMA References

Aspect-Orientation

[1] J. Kienzle, Ekwa Duala-Ekoko and S. Gélineau, "AspectOPTIMA:A Case Study on Aspect Dependencies and Interactions", Transactions on Aspect-Oriented Software Development, in press.

[2] J. Kienzle and S. Gélineau, " AO Challenge: Implementing the ACID Properties for Transactional Objects", in Proceedings of the 5th International Conference on Aspect-Oriented Software Development - AOSD 2006, March 20 - 24, 2006, pp. 202 – 213, ACM Press, March 2006.

[3] J. Kienzle and R. Guerraoui, " AOP - Does It Make Sense? The Case of Concurrency and Failures", in 16th European Conference on Object–Oriented Programming (ECOOP'2002), Lecture Notes in Computer Science 2374, (Malaga, Spain), pp. 37 – 61, Springer Verlag, 2002.

Open Multithreaded Transactions

[4] M. Monod, J. Kienzle, and A. Romanovsky, "Looking Ahead in Open Multithreaded Transactions", in Proceedings of the 9th International Symposium on Object and Component-Oriented Real-Time Distributed Computing, pp. 53 – 63, IEEE Press, April 2006.

[5] J. Kienzle, Open Multithreaded Transactions — A Transaction Model for Concurrent Object-Oriented Programming. Kluwer Academic Publishers, 2003.

McGill