

JavaD: Bringing Ownership Domains to Mainstream Java

Marwan Abi-Antoun
Ph.D. Project Presentation
17-754: Analysis of Software Artifacts

Some slides adapted from a talk by
Neelakantan Krishnaswami



Analysis of Software Artifacts -
Spring 2006

Why Ownership Domains?



- “The big lie of object-oriented programming is that objects provide encapsulation” (Hogg)
- Aliasing can cause a failure of encapsulation

```
class JavaClass {
    private List signers;

    public List getSigners() {
        return this.signers;
    }
}

// (Malicious) clients can mutate signers field!
class MaliciousClient extends ... {
    public void addTrojanHorse(JavaClass c)
    {
        List signers = c.getSigners();
        signers.add( this );
    }
}
```

Analysis of Software Artifacts -
Spring 2006

2

Aliasing is a necessary evil



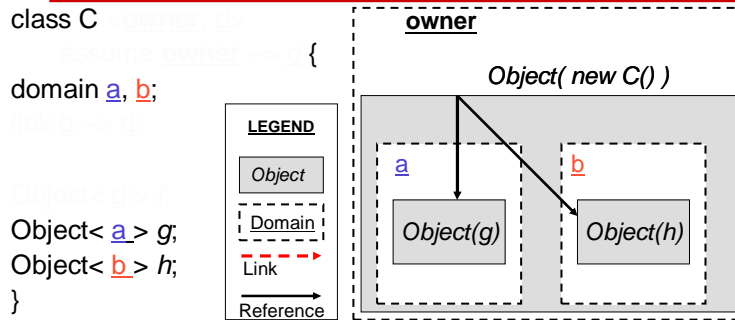
- Aliasing cannot be eliminated
 - Object-oriented design patterns rely on it
- Aliasing can/must be controlled
 - Need for language support for this
- Several solutions proposed
 - Ownership Domains (AliasJava)
- Many paper-only designs
 - AliasJava notable exception
- Few evaluation on large case studies
 - AliasJava, Universes case studies

Ownership Domains Defined



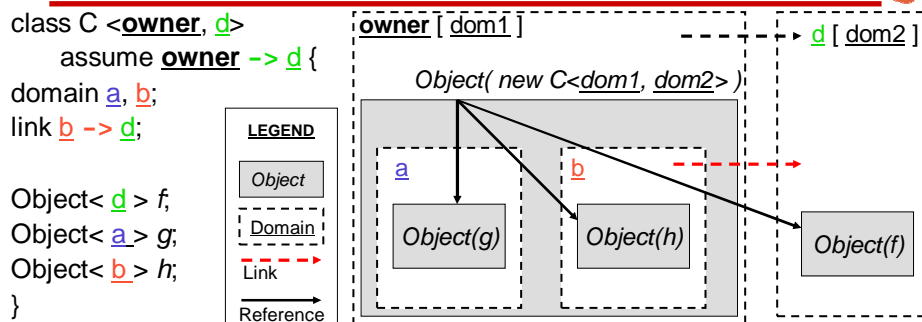
- “Object ownership (instance encapsulation) ensures that objects cannot be leaked beyond an object or collection of objects which own them” (Alex Potanin)
- Ownership domain = region of the heap
- How does it control aliasing?
 - Within a given domain, there *can be* aliasing
 - *No* aliasing between two given domains
 - *Explicit* permissions for cross-domain access (creation, reference, etc)

Ownership domains



- Every object is in exactly one domain
- Every object can have one or more domains
 - Domains a and b are declared in class C
 - *Object*<a> *g* means object *g* is in domain a

Ownership domain parameters



- Domain parameters use syntax similar to type parameters
 - d is a domain parameter
- Link declarations specify that objects in domain b have permission to access objects in domain d

AliasJava (by Aldrich et al.)



- Concrete implementation of Ownership Domains
 - Language extension to Java (Barat infrastructure)
 - Basic tool support (no debugger!)
- Keyword **domain** define ownership domains
- Java 1.5 type parameters syntax to define
 - Domain parameters: `class Sequence<Towner>`
 - Binding actuals to formal: `Sequence<owned> seq;`
- Aliasing annotations describe data:
 - Confined with object (“**owned**”) (default domain)
 - Passed linearly from one object to another (“**unique**”)
 - Shared temporarily (“**lent**”) within method
 - Shared persistently (“**shared**”) globally

Signers Example in AliasJava



```
class JavaClass {
    private owned List signers;

    private owned List getSigners() {
        return this.signers; }

    public void foo() {
        lent List x = this.getSigners();
        // do stuff using x
    }
}
```

- **owned** default private domain on each object
- Clients cannot invoke `getSigners()` since objects outside of `JavaClass` cannot access `JavaClass`'s **owned** domain
- Clients can only invoke `foo()`

Signers Example in AliasJava



```
class JavaClass {
    private owned List signers;

    public shared List getSigners() {
        shared List copy = new List();

        for(int i = 0; I < this.signers.size();i++)
            copy.add(this.signers.get(i));
        return copy;
    }
}
```

- Making `getSigners()` return a globally shared copy

JavaD: AliasJava with annotations



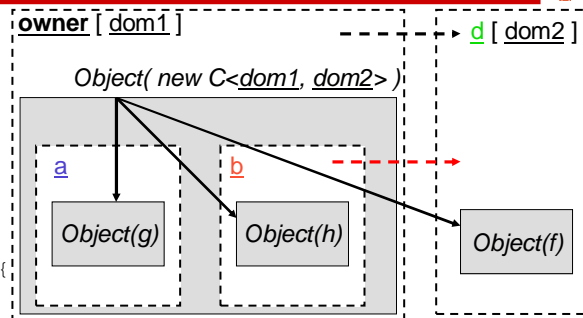
- Use annotation facility in Java 1.5
 - No language extension
- Use Eclipse JDT infrastructure + Crystal
 - Much improved tool support!
 - Debugging, refactoring, syntax highlighting, ...
- Make it easier to add features to the language
 - External uniqueness, read-only references, ...
- Incrementally and partially specify annotations
 - Necessary for dealing with large code bases
- Usability
 - Generate **warnings** about inconsistent annotations
 - Supply reasonable defaults

JavaD: ownership domain annotations



```
@Domains({"a", "b"})
@DomainParams({"d"})
@DomainLinks({"b->d"})
@DomainInherits({"IC<d>"})
public class C implements IC {
    @Domain("d") B f = new B();
    @Domain("a") B g = new B();
    @Domain("b") B h = new B();
}

@DomainParams({"d"})
public interface IC implements IC {
}
```



@Domains: declare domains

@DomainParams: declare *formal* domain parameters

@DomainLinks: declare domain link specifications

@DomainInherits: specify parameters for superclass/interfaces

@Domain: specify object domain and specify *actual* domain parameters

@DomainReceiver: specify annotation for constructor/method receiver

Analysis of Software Artifacts -
Spring 2006

11

Tool Design and Implementation



- Annotation information management
 - Retrieve annotations from AST
 - Parse annotation values
- First Pass (visitor-based analysis)
 - Identify problematic code patterns
 - Propagate local annotations
 - Map AST nodes to annotations
- Second Pass (visitor-based analysis)
 - Check annotations using AliasJava rules
 - Intra-procedural live variables analysis

Analysis of Software Artifacts -
Spring 2006

12

Annotation Information



- For each AST node, maintain
 - Annotation (e.g., “lent”)
 - Parameters
 - ArrayParameters
 - Map from Formals to Actuals
- Work around Java annotation limitations
 - Only use `@Target({ElementType.PARAMETER, ...})` to specify where annotation is allowed
 - Otherwise, use free form string annotation value
- JavaCC for parsing annotations
 - “parameter <parameter, ...> [arrayParameter, ...]”
 - “obj.dom <dom_{i1}, ..., dom_{in}> [dom_{j1}, ..., dom_{jn}]”

Identify Problematic Patterns



- Replace with equivalent constructs
 - Declare a local variable (built-in refactoring)
 - Add appropriate annotations

- **New Expressions**

```
public Iterator getIter() {  
    return new SequenceIterator(head);  
}
```

- **Cast Expressions**

```
ArrayList vCourse = objStudent.getRegisteredCourses();  
for (int i=0; i<vCourse.size(); i++) {  
    if (((Course) vCourse.get(i)).conflicts(objCourse)) {  
        lock.releaseLock();  
        return "Registration conflicts";  
    }  
}
```

Propagate Local Annotations



- AST Visitor
- Read annotations from
 - TypeDeclarations
 - Variable/Field Declarations
 - Method Declarations
- Translate Formals to Actuals
- Infer default annotations
 - Unique on NullLiteral, StringLiterals
- Map ASTNode to annotation
 - Used by the second pass

Check annotations



- AST Visitor to implement AliasJava rules
 - TypeDeclaration: inheritance rules
 - FieldDeclaration: declaration rules
 - SingleVariableDeclaration: declaration rules
 - VariableDeclarationFragment: declaration rules
 - MethodDeclaration: check method rules
 - Assignment: check assignment, initializers
 - ClassInstanceCreation: constructor rules
 - MethodInvocation: method call rules
 - ReturnStatement: assignment
 - FieldAccess: assignment

Check Method Declaration

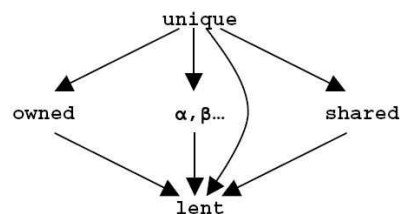


- Check Return Type Annotation
 - If reference type, must have an annotation
 - May not be marked **owned** for public methods
- Check Parameter Annotations
 - If reference type, must have an annotation
 - May not be marked **owned** for public methods
- Check Overriding
 - May not change return type annotation
 - May not change parameter annotation
 - May not change receiver annotation
- Each annotation must have appropriate binding from actuals to formals

Value flow analysis



- Checking assignment
- Value flow analysis
 - Not dataflow analysis
 - Arrow means data can flow between variables with two annotations
- **Live variables analysis** to check “destructive read”
 - Data flow analysis
 - Reused from Crystal



- Variable with any type annotation can be assigned a **unique** value
- **lent** variables can be assigned a value with any type annotation
- Values with type annotations **owned** and **shared**, as well as declared domains kept separate from each other

Lessons Learned



- Java 1.5 annotations too limiting
 - @owned vs. @Domain("owned") or @Domain("owned <owned>")
- Restrictions on certain coding constructs
- Annotations too verbose
 - Combine ownership and generic types (Potanin et al.)
 - Consider for example a box as a kind of object
 - Plain OO: "this is a box"
 - Generics: "this is a box of books"
 - Ownership: "this is my box", "these are library books"
 - Ownership + generics: "this is my box of library books"

Limitations and Future Work



- Support adding annotations to JDK and other third-party libraries
 - Place annotations in separate files
- Additional case studies
- Develop new kinds of annotations
 - "extunique", "readonly", ...
 - @Ignore, @Suggest, @Complete
- Make it easier to infer annotations interactively
 - Use Eclipse preview refactoring functionality
 - Annotating existing code difficult
 - Determining ownership parameters
 - Annotating existing code time-consuming
 - Every line of code with a reference type!

Questions?



References



- Aldrich, J. and Chambers, C. Ownership Domains: Separating Aliasing Policy from Mechanism. In ECOOP, 2004.