

# Demeter Interfaces: Adaptive Programming without Surprises

Therapon Skotiniotis

Jeffrey Palm

Karl Lieberherr

College of Computer & Information Science  
Northeastern University  
360 Huntington Avenue  
Boston, Massachusetts 02115 USA  
{skotthe,jpalm,lieber}@ccs.neu.edu

## ABSTRACT

Adaptive Programming (AP) allows for the separate definition of data structures, traversals and computations over these data structures. The loosely defined contexts, structure and computation, are composed according to a given traversal specification. Traversal specifications are defined against a graph-based model of the underlying data structure with the ability to abstract over graph node names, edges and subpaths. As such certain alterations to the data structure will not affect the meaning of the program. Currently AP systems, *e.g.* DemeterJ, provide no mechanisms to warn or even guard against modifications that *will* affect the meaning of a program. Programmers have to depend on thorough testing in order to detect such modifications.

In this paper we present *Demeter Interfaces*, an extension to the DemeterJ system. Demeter Interfaces specify the expected structural properties of the underlying data structure that must hold in order for adaptive code to function correctly. Through an example we illustrate the usage and implementation of Demeter Interfaces. We further illustrate how Demeter Interfaces result in better designs of adaptive programs, ease of adaptive code reuse and how they promote parallel development. We discuss how Demeter Interfaces are applicable to XML programs that use the XPath navigation operator *//*. This makes our results relevant to large communities, such as the XQuery and XSLT communities.

## 1. INTRODUCTION

An adaptive program is a program written in terms of loosely coupled contexts, *i.e.*, data structure and behavior (computations) with a third definition succinctly binding the two contexts together. In DemeterJ [8], the most popular incarnation of AP, programs are defined in terms of their data structures and a set of computations that take place during the traversal of data structures. Structure and computation come together through *traversal specifications*. Traversal specifications can abstract over graph node names, edges and subpaths that allows certain modifications to the underlying data structure without altering the meaning of the whole computa-

tion. Traversal specifications are defined using a domain specific language that operates on a graph-based model of a program's data structure. Identifying which modifications do not alter the meaning and intent of the computation is not a trivial task. Identifying which parts of the data structure are explicitly referenced in traversal specification and their attached computations is not enough. Assumptions made by the adaptive code about the underlying data structure have to be verified. As a simple example consider a traversal specification used to obtain an object deep inside a data structure, *e.g.*, "*from Company to CEO*", on a data structure representing a company. If this traversal specification was written with the assumption that there is only **one** CEO in a company and a modification is made to allow multiple CEOs then no compile time error is issued by DemeterJ. The program compiles and executes, however the result of the computation will be incorrect. DemeterJ offers no way to define and check for such assumptions. Programmers have to rely on extensive tests that need to be run every time a modification is made, defeating some of the benefits of adaptive programming.

The problems of modifications that alter the meaning of the program makes iterative and parallel development difficult. As dependencies between computations and traversals arise, it becomes harder to properly test and detect bugs in adaptive programs. At the same time, as AP programs become larger in size understandability of adaptive code decreases. Programmers have to both look at the traversal specification and the computation attached to it but also calculate from the concrete data structure the possible paths based on the traversal specification. Finally, since both traversal specifications and their attached computations refer to concrete data structure names we get a decrease in reusability of adaptive code.

In this paper we introduce Demeter interfaces that give us AP without surprises. Programs are defined through enhancements (aspects) to a high-level traversal strategy written against a graph based model of the underlying data structure. If the data structure changes, the program might still work without any changes to the strategy nor the aspects that enhance the program. This gives adaptive programs an impressive flexibility to data type changes. But this flexibility is also dangerous; for the new data type, the program might work but give the wrong result. Demeter interfaces are intended to restrict the set of data types that are allowed to be used with an adaptive program.

Demeter Interfaces hit a sweet spot between flexibility and safety. They restrict a bit what AP can do but without going back to the old way of writing the Structural Recursion template manually [9]. They are safer because the adaptive programs intent is defined and used to check any future data type against it. As a side effect, adaptive programs become better documented and more understandable.

Demeter Interfaces (DIs) as a mechanism within DemeterJ allow programmers to define:

- traversal strategies,
- an *Interface Class Graph (icg)*, which provides an interface for the concrete data structure
- a *Visitor Interface*, used to implement the attached computation
- a set of *Constraints*, that define properties that both the icg and the underlying data structure must satisfy

The concrete data structure definition is extended to accommodate for a name mapping between its data members and those of the DI that it implements. The DemeterJ system is extended to statically verify the mapping and validate all DI constraints.

Section 2 introduces Demeter Interfaces by first presenting an example application implemented in DemeterJ first without and then with Demeter Interfaces. Section 3 discussed the design benefits enjoyed by adaptive programs that deploy DIs. Section 4 discusses related work and section 5 presents future work. Section 6 concludes.

## 2. DEMETER INTERFACES

Demeter Interfaces provide a view of an adaptive program that encloses all the necessary assumptions and structure properties of the underlying concrete data structure, traversal specifications and a visitor interface. Computation is attached to the traversal via the implementation of the visitor interface.

The rest of this section illustrates the usage of DIs and their advantages through an equation system example and the implementation of a semantic checker. We first provide a solution in DemeterJ [8] which we also use to describe the current DemeterJ system. We then iteratively extend the equation system, exposing the issues with the current DemeterJ implementation. We then show a solution that uses DIs and analyze the advantages over our initial implementation.

### 2.1 A simple equation system in DemeterJ

Our example is about systems of equations in which we want to check that all used variables are defined (we call this a *semantic checker*). A simple equation system could be  $x = 5; y = 9; z = x + y;$ .

Adaptive programs in DemeterJ are defined through a Class Dictionary (**cd**), a set of Behavior Files (**beh**) and Java code. Listing 1 and Listing 3 make up the DemeterJ code for the simple equation system and the implementation of the semantic checker that prints out two lists; defined variables and used variables.

Listing 1: Class Dictionary for Simple Equations

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> Variable "=" <rhs> Expr.
Expr : Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr)".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".
visitor DefCollector = .
```

A cd file is a textual representation of the Object-Oriented structure of the program which specifies classes and their members. Figure 1 shows the UML representation of the class dictionary in Listing 1. Each line of the class dictionary defines a class definition. An equal sign (“=”) defines a concrete class with the class name on the left hand side of the equals and the members of the class in the right hand side of the equals. Names enclosed in < > define class member variable names that represent edges in the graph representation of the class graph. Replacing the equal sign with a colon (“:”) defines an abstract class with its subclasses on the left of the colon. Text within left and right angle braces defines class member variable names. The DemeterJ system uses a class dictionary as a grammar definition, providing a language that can parse in definitions and create the appropriate objects. Tokens in the class dictionary surrounded in quotes define the generated languages’ syntax tokens (Listing 2). Parameterized classes are defined through the tilde (“~”) operator. The **visitor** keyword specifies a visitor class.<sup>1</sup>

Listing 2: An instance of a simple equations system given as input to DemeterJ

```
(
  x = 5;
  y = (x - 2);
  z = ((y-x) + (y+9))
)
```

Behavior files are used to introduce code to the classes specified in the accompanying class dictionary. Introductions can be in the form of regular Java methods, adaptive methods or visitor methods. In Listing 3, the definition of `main` is introduced into the class `Main` as a regular Java method.

Three adaptive method signatures are introduced into `EqSystem`, called `print`, `printDefined` and `printUsed`. Adaptive method signatures are similar to normal Java method signatures extended with a traversal specification, e.g., `to *`, and passed the name of a Visitor class.<sup>2</sup> The adaptive method `printUsed` will introduce a method into `EqSystem` that takes no arguments and returns `void`. When called, the method will traverse from an instance of `EqSystem` to instances of `Ident`. The traversal is not allowed to go through any edge with the label `lhs`. At every instance that the traversal comes across the `DefCollector` visitor definition is advised. If there is a visitor directive, e.g., **before**, **after** or **around**, and the directives argument matches the instance type, the code attached to that directive is executed. Before traversing a `Variable` instance the traversal will print out its contents.<sup>3</sup>

Listing 3: Behavior File for Simple Equations

```
Main {
  public static void main(String args[]) throws Exception { {
    EqSystem eqSystem = EqSystem.parse(System.in);
    System.out.println("Input is : ");
    eqSystem.print();
    System.out.println("");
    System.out.println("Defined are : ");
    eqSystem.printDefined();
    System.out.println("");

    System.out.println("Used are : ");
    eqSystem.printUsed();
    System.out.println("");
  } }
}
```

<sup>1</sup>The `DefCollector` class has no data members.

<sup>2</sup>The `PrintVisitor` prints all instances in a traversal and it is automatically generated by the system

<sup>3</sup>`Ident` is a DemeterJ wrapper class for Java’s `String` class.

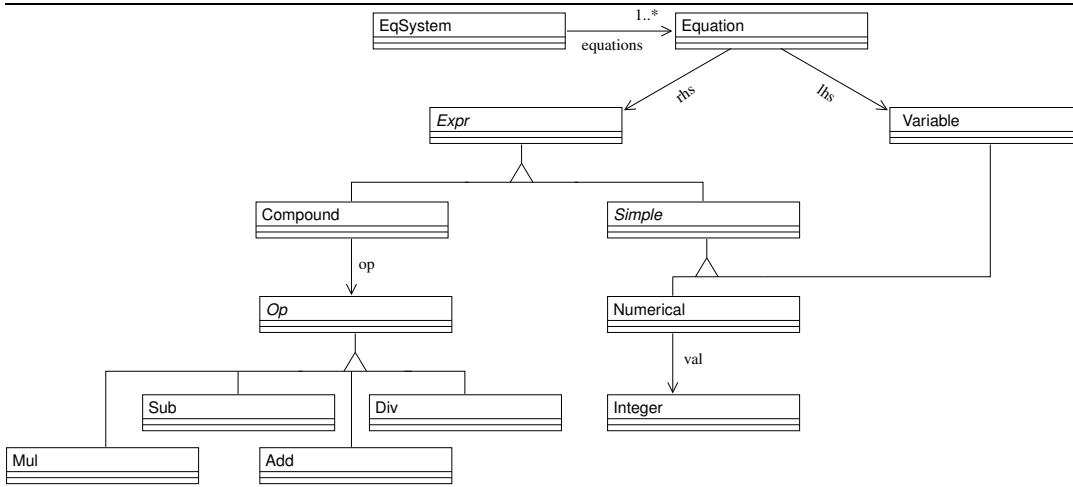


Figure 1: The UML equivalent of Simple Equations Class Graph.

```

}
EqSystem {
  public void print() to * (PrintVisitor);

  public void printDefined() bypassing -> *,rhs,* to Ident (
    DefCollector);

  public void printUsed() bypassing -> *,lhs,* to Ident (
    DefCollector);
}
DefCollector {
  before Variable {{
    System.out.println("Found Ident : " + host.get_ident());
  }}
}

```

With the completed AP implementation of the semantic checker in place we can now evaluate our solution and verify the claims made, both in favor and against, AP.

The principle behind AP states [6]

“A program should be designed so that the interface of objects can be changed within certain constraints without affecting the program *at all*.”

For example, altering the equation system so that equations are now in prefix notation requires a single modification to `Compound`'s definition in the class dictionary:

```

Compound = ``('' <op> Op <lrand> Expr
  <rrand> Expr ``)``.

```

Leaving the rest of the code intact the behavior of the program remains the same. The modification did not alter any of the traversed objects that are used to implement the semantic checker.

However any alteration that either

- modifies class and/or class member variable names that are explicitly referenced by traversal and/or visitors,
- or, breaks an assumption about the system on which an adaptive method depends

alters the program's behavior. For example, altering the equations system to allow for function definitions with arguments causes the

no compile error, but results in erroneous program behavior. This modification breaks two assumptions

1. There is only one `Variable` defined at each equation.
2. All variables have global scope and thus can be used anywhere.

The adaptive methods, as well as the visitor, depend on these assumptions. However these assumptions are not explicitly captured in AP programs. There is no tool support to stop such modifications. In fact naively extending the equation system to accommodate for functions parameters (Listing 4) will generate a valid AP program with the wrong behavior for the semantic checker.

With larger AP programs, it becomes nearly impossible to find all these implicit assumptions and even harder predict which modifications will cause erroneous behavior. Programmers have to rely on exhaustive testing in order to increase their confidence that the program still behaves according to its specification.

Listing 4: Class Graph for equations systems with functions of one argument.

```

EquationSystem = <equations> BracketList(Equation).
Equation = <lhs> VarOrFunc "=" <rhs> Expression.
VarOrFunc : DVariable | Function.
Function = "fun" Variable "(" CommaList(Variable) ")".
Expression : FunCall | Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
DVariable = Ident.
FunCall = Variable "(" CommaList(Simple) ")".
Numerical = <val> Integer.
Compound = "(" <lrand> Expression <op> Op <rrand>
  Expression ")".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BracketList(S) ~ "(" S {"," S} ")".
CommaList(S) ~ S {"," S}.

```

Addressing these issues requires

- The ability to define assumptions about the underlying data structure,

- Tool support to allow the validation of these assumptions,
- Decrease the dependency on class and class member variable names,
- The modularization of only the relevant program information for each adaptive behavior instead of the whole class graph.

## 2.2 A simple equation system with Demeter Interfaces

A Demeter Interface resides between a class graph and the *implementation* of adaptive behavior, *i.e.*, adaptive methods and visitor implementations. A DI is made up of

- an interface class graph [7] (**icg**) that defines a portion of a program structure to which adaptive behavior will be attached,
- traversal strategies that act on the icg,
- constraints imposed on the icg and the traversal strategies,
- adaptive method signatures,
- and a visitor interface.

Figure 2 shows the Demeter Interface for the simple equation system along with a diagrammatical representation (in UML) of the DI's interface class graph.

The **icg** serves as an abstraction of any class graph implementation of the ExprICG DI in order for the strategies defined in ExprICG to be applicable. Furthermore, the constraints defined in the Demeter Interface impose further requirements that need to be met by any concrete implementation of this Demeter Interface. The first constraint expresses that each variable definition introduces exactly one name. The non-empty constraints state that there must be at least one path in the class graph that satisfies the strategy given as parameter. The predicates *unique* and *nonempty* are provided by DemeterJ. The **visitor** section is a list of Java interface definitions for the visitor that need to be implemented for this DI. Method names *before*, *after* and *around* will have the same meaning as the visitor directives in DemeterJ. The **adaptive methods** section holds a list of adaptive method signature introductions. Adaptive methods now take the visitor as an argument to the method.

Figure 3 gives an example implementation of the ExprICG Demeter Interface (on the right) along with a visitor implementation and a driver class (on the left). InfixEQSystem defines the structural relations between concrete classes.

DemeterJ's class dictionaries are extended in two ways; a header is introduced allowing for an **implements** statement that specifies which DIs are being implemented and a mapping between the concrete class dictionary classes to the icg's classes. The concrete class dictionary InfixEQSystem provides a definition of its equation system and a mapping  $M$  between the entities in its class graph and the entities in the interface class graph. The mapping definition can map class(es) to class(es), class member variable name(s) to class member variable name(s), strategy to strategy and class to strategy. Invoking the adaptive method requires that a concrete visitor that implements DVisitor is given as an argument. The addition of DI requires modifications to the compile time behavior of DemeterJ. For each concrete class dictionary that implements a Demeter Interface

1. Use the DI's mapping  $M$  to ensure that the concrete class dictionary (InfixEQSystem) satisfies the interface class graph from the DI (ExprICG), *i.e.*, for each edge  $e$  connecting two nodes  $n_1$  and  $n_2$  in the interface class graph

the mapped edge  $M(e)$  connects the mapped nodes  $M(n_1)$  and  $M(n_2)$  in the class dictionary. The mapping has to map each entity of the icg to some (possibly many) entity in the class dictionary. The tool does support some limited form of inference. After reading in the explicit mapping definitions, the tool follows the structure of the icg and the concrete class graph and attempts to map the remaining entries. The mapping occurs by position for concrete classes and by expanding the definition of abstract classes. If a mapping is not possible an error is reported. For example, in the case of InfixEQSystem, EquationSystem is mapped to ESystem that also maps Definition to Equation. For the mapping of Body the system expands the choices for Expression which leads to an element (Variable or Numerical) or a list of Expressions and Op. The system takes the alternatives of an Expression class and maps those to Body.

2. Strategies defined in the DI will be expanded according to the mapping and made available through InfixEQSystem class.
3. Each of the constraints will be validated on the expanded strategies for InfixEQSystem.
4. For each visitor method (*before*, *after* and *around*) the method's argument type has to part of the icg. Also, any visitor implementation must have the **exact** *before*, *after* and *around* methods as its interface. If that is not the case, the system signals an error that the visitor implementation introduces an extra dependency that is not allowed but the DI.

If any of the above steps fails then the concrete class dictionary failed to implement its DI interface. With the simple equation system implemented using Demeter Interfaces we now extend the system and verify that DIs assist the prevention of modifications that alter the program's behavior.

As a first evolution step we want to change from infix notation to prefix notation. This is a modification that does not alter the program's behavior even in the original DemeterJ solution without DIs. Moving to a prefix notation requires to change the definition of Compound in InfixEQSystem to

```
Compound = <op> Op <lrand> List(Expression)
           <rrand> List(Expression)
```

This change does not affect the Demeter Interface at all. We update the equation system class graph and keep the existing mapping. All constraints of the DI are still satisfied after they are mapped into the actual interface class graph and the adaptive methods will function correctly.

It is important to note that during this evolution step, only the DI and the concrete implementation of the interface class graph was needed. Furthermore, the static assurances provided by the tool because of the DI, did not require re-testing of the driver code to ensure that the strategies pick the correct traversals and that the semantic checker still operates as expected. The Demeter Interface allows in this case for separate development and ease of evolution. Hiding irrelevant information through the Demeter Interface provides for higher system modularity.

In the next evolution step we want to add functions with arguments to the equation system. This evolution step affects information that is relevant to the semantic checker. The semantic checker has to also deal with parameter names on each function definition

```

di ExprICG {
  icg:
    ESystem = List(Definition).
    Definition = <def> DThing "=" <body> Body.
    Body = List(UThing).
    UThing = Thing.
    DThing = Thing.
    Thing = Ident.
    List(S) ~ "(" S ") ".

  strategies:
    gdefinedIdents = "from ESystem via DThing to Ident".
    gusedIdents = "from ESystem via UThing to Ident".
    definedIdent = "from Definition via DThing to Ident".

  constraints:
    unique(definedIdent).
    nonempty(gusedIdents).
    nonempty(gdefinedIdents).

  visitors:
    interface DVisitor {
      public void before (Thing t);
    }

  adaptive methods:
    ESystem {
      public void printDefined(DVisitor) gdefinedIdents;
      public void printUsed(DVisitor) gusedIdents;
    }
}

```

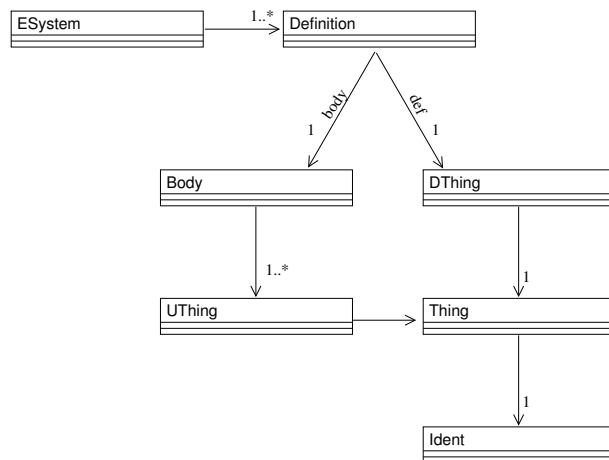


Figure 2: The full definition of the Demeter Interface includes an interface class graph, the strategies on the interface class graph and constraints on the strategies. The UML diagram is equivalent to the interface class graph defined in `ExprICG`.

but also usages of function definitions that may appear on the right-hand side of equations. Function parameters do not have global scope, their scope is local to the function definition. A naive approach would be to alter the class dictionary to accommodate for function definitions (Listing 5).

Listing 5: Extending the class dictionary to accommodate function definitions by changing the class dictionary only.

```

cd ParamEquations implements ParamExprICG {
  EquationSystem = <equations> BracketList(Equation).
  Equation = <lhs> VarOrFunc "=" <rhs> Expression.
  VarOrFunc : Variable | Function.
  Function = <fname> Variable
    "(" <args> CommaList(Variable) ")".
  Expression : FunCall | Simple | Compound .
  Simple : Variable | Numerical.
  Variable = Ident.
  FunCall = <fname> Variable
    "(" <fargs> CommaList(Simple) ")".
  Numerical = <val> Integer.
  Compound = "(" <lrand> Expression <op> Op
    <rland> Expression ")".
  Op : Add | Sub | Mul | Div.
  Add = "+".
  Sub = "-".
  Mul = "*".
  Div = "/".
  BracketList(S) ~ "(" S {"," S} ")".
  CommaList(S) ~ S {"," S}.
}

```

```

use EquationSystem as ESystem,
  (->*,lhs,* to Variable) as DThing,
  (->*,rhs,* to Variable) as UThing,
  Variable as Thing.
}

```

Without altering the mapping and leaving the Demeter Interface intact this approach will result in a compile time error. The mapping definition does not raise an error since `DThing` can be mapped to `Variable` by going either through `DExpression` or through `ParametricVariable`. The predicate `unique` for the traversal specification of `definedIdent`, however, can no longer be satisfied. The modification to allow functions with one parameter breaks one of the assumptions of the interface in particular the fact that we can reach more than one variable through the left hand side of the equal sign. With one argument functions the meaning of what is defined and what is its scope has changed and these changes have to be reflected in the Demeter Interface.

It is essential that for this evolution step the interface has to change (Figure 4). With a new interface class graph `ParamExprICG` we can abstractly reason about semantically checking systems with functions. The strategies `definedIdent` and `usedIdents` are used to navigate to definitions and usage of variable names, both function names as well as simple variables. The strategies `dName` and `uName` are then used to collect arguments (at function definition) and formal arguments (at function invocation) respectively. The visitor interface defines the method `return` which is called by `DemeterJ` at the end of a traversal. The return value of the `return` method is also the return value of the traversal. The implementa-

<pre> class Main {   public static void main(String[] args){     DVisitor v = new IDPrintVisitor();     ParamEQSystem ieqs = InfixEQSystem.parse(System.in);     System.out.println("IDs in def:");     ieqs.printDefined(v);     System.out.println("IDs in use:");     ieqs.printUsed(v);   } }  class IDPrintVisitor extends DVisitor {   public void before(Thing id) {     System.out.println(id.toString());   } } </pre>	<pre> cd InfixEQSystem implements ExprICG {   EquationSystem = &lt;equations&gt; List(Equation).   List(S) ~ "(" {S} ")".   Equation = &lt;lhs&gt; Variable "=" &lt;rhs&gt; Expression.   Expression : Simple   Compound.   Simple : Variable   Numerical.   Variable = Ident.   Numerical = &lt;v&gt; int.   Compound = &lt;lrand&gt;List(Expression) &lt;op&gt; Op &lt;rrand&gt;     &gt;List(Expression).   Op : Add.   Add = "+".    use EquationSystem as ESystem,     (-&gt;*,lhs,* to Variable) as DThing,     (-&gt;*,rhs,* to Variable) as UThing,     Variable as Thing. } </pre>
---	---

Figure 3: `InfixEQSystem` defines a class graph and a mapping of the entities in the class graph to the interface class graph of `ExprICG`. The driver class `Main` uses the strategies defined in `ExprICG` and the `IDPrintVisitor`.

tion of method `checkBindings` is introduced into `ESystem` and it is used to check the correct usage of variable and function definitions. The inputs to these functions are two lists where the first argument represents variable and function definition at different scopes and the second represents variable and function usage at their corresponding scope.

Listing 6 shows the class graph that implements `ParamExprICG`. The class graph maps the edge `args` to the edge `fparam` and its source and target nodes accordingly. While the other edge mapping maps the edge `fargs` to the edge `aparam` but the targets of `fargs` are mapped to the `Variable` node that can be reached from the `aparam` edge. Figure 5 shows the visitor implementation and the driver class.

Listing 6: Modifications to the concrete class dictionary to accommodate single argument functions.

```

cd ParamEquations implements ParamExprICG{
  EquationSystem = <equations> BracketList(Equation).
  Equation = <lhs> VarOrFunc "=" <rhs> Expression.
  VarOrFunc : Variable | Function.
  Function = <fname> Variable
    "(" <args> CommaList(Variable) ")".
  Expression : FunCall | Simple | Compound .
  Simple : Variable | Numerical.
  Variable = Ident.
  FunCall = <fname> Variable
    "(" <fargs> CommaList(Simple) ")".
  Numerical = <val> Integer.
  Compound = "(" <lrand> Expression <op> Op
    <rrand> Expression ")".
  Op : Add | Sub | Mul | Div.
  Add = "+".
  Sub = "-".
  Mul = "*".
  Div = "/".
  BracketList(S) ~ "(" S {" S } ")".
  CommaList(S) ~ S {" , " S}.

  use EquationSystem as ESystem,
    (->*,args,*) as (->*,fparam,*),
    (->*,fargs,*) as (->*,aparam,*) to Variable,
    Variable as Thing.
}

```

In this evolution step, the Demeter Interface helped by disallow-

ing a naive extension that would violate the intended behavior of the original Demeter Interface. The nature of the evolution required an extension of the interface and that resulted to changes in the driver class and a new concrete class dictionary. It is important to note how the Demeter Interface exposed the erroneous usage of the `ExprICG` interface for this evolution step and assisted in updating all the dependent components due to the definition of `ParamExprICG`.

## 2.3 Demeter Interfaces and XPath

Ideas in AP can be found in other technologies where the separation between navigation code and computation is necessary. According to the abstractions that traversal specifications allow, the problems of surprise behavior are present in these systems as well. XML and XPath queries are technologies widely used today that share similar issues with AP. Specifically one can think of DTD as class dictionaries and XPath expression as traversal strategies. The problems of surprise behavior are prominent in these technologies as well since modifications to the XML document might break assumptions that the XPath query depends upon. Consider the following DTD

```

<?xml version="1.0"?>
<!ELEMENT BusRoute ((DieselBus|GasBus)*,Town*)>
<!ELEMENT DieselBus (Person*)>
<!ELEMENT GasBus (Person*)>
<!ELEMENT Town (BusStop*)>
<!ELEMENT BusStop (Person*)>
<!ELEMENT Person EMPTY>
<!ATTLIST BusRoute name CDATA #REQUIRED>
<!ATTLIST DieselBus name CDATA #REQUIRED>
<!ATTLIST GasBus name CDATA #REQUIRED>
<!ATTLIST Town name CDATA #REQUIRED>
<!ATTLIST BusStop name CDATA #REQUIRED>
<!ATTLIST Person name CDATA #REQUIRED>

```

and the JavaScript statement containing the XPath expression

```

var nodes=xmlDoc.selectNodes("./BusStop//Person
| ./DieselBus//Person")

```

The XML document above, along with the XPath query, aim at selecting the people that are either waiting at a bus stop or riding in a diesel bus. Now consider the following modification to the original DTD

```

di ParamExprICG {
  icg:
    ESystem = List(Definition).
    Definition = <def> DThing "=" <body> Body.
    Body = List(UThing).
    UThing = Thing [ " (" <aparam> UParamName " ) " ].
    DThing = Thing [ " (" <fparam> DParamName " ) " ].
    Thing = Ident.
    UParamName = Ident.
    DParamName = Ident.
    List(S) ~ " ( " S " ) ".

  strategies:
    definedIdents = "from ESystem to DThing".
    usedIdents = "from ESystem to UThing".
    dName = "from DThing via → *,fparam, * to Thing".
    uName = "from UThing via → *,aparam, * to Thing".

  constraints:
    nonempty(definedIdents).
    nonempty(usedIdents).

  visitors:
    interface PVisitor {
      public void before (Thing t);
      public LinkedList return ();
      public void before (DThing t);
      public void before (UThing t);
    }

  adaptive methods:
    ESystem {{
      public LinkedList printDefined(PVisitor) definedIdents;
      public LinkedList printUsed(PVisitor) usedIdents;
      public boolean checkBindngs(LinkedList l1, LinkedList l2){
        // checks appropriate usage of variables
      }
    }}
    DThing {{
      public LinkedList getDefined(PVisitor) dName;
    }}
    UThing {{
      public LinkedList getUsed(PVisitor) uName;
    }}
}

```

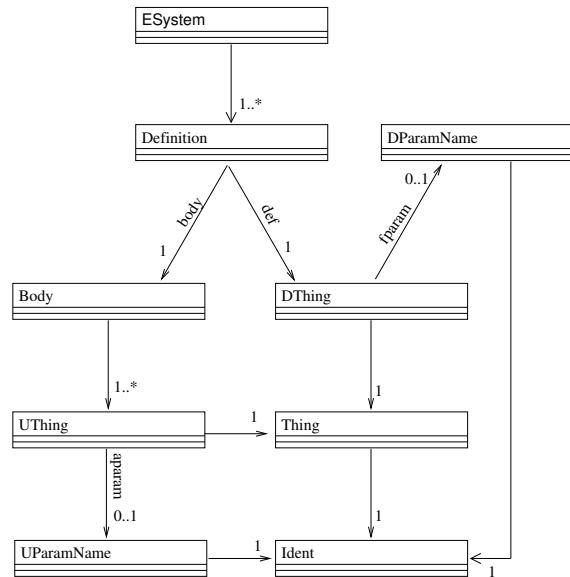


Figure 4: The evolved Demeter Interface (left) and the UML representation of the extended Demeter Interface class graph (right).

```

class Main {
    public static void main(String[] args){
        GVisitor defV = new GVisitor();
        GVisitor useV = new GVisitor();
        ParamEquations pe = ParamEquations.parse(System.in);
        boolean codeOk = pe.checkBindings(pe.printDefined(
            defV), pe.printUsed(useV));
        if (!codeOk)
            System.out.println(" Variables used before they where
                defined");
    }
}

```

```

import java.util.LinkedList;

class PVisitor {
    LinkedList env;

    PVisitor(){
        this.env = new LinkedList();
    }
    public void before(Thing t) {
        env.add(t);
    }
    public void before(UThing ud) {
        LinkedList rib = getUsed(this);
        env.add(rib);
    }
    public void before(DThing ud) {
        LinkedList rib = getDefined(this);
        env.add(rib);
    }

    public LinkedList return(){
        return env;
    }
}

```

Figure 5: Changes to the interface affect Main. The definition of GVisitor is used to check for the local parameter names in parametric equations.

```

<?xml version="1.0"?>
<!ELEMENT BusRoute ((DieselBus|GasBus)*,Town*)>
<!ELEMENT DieselBus (Person*)>
<!ELEMENT GasBus (Person*)>
<!ELEMENT Town (BusStop*)>

<!ELEMENT BusStop (Person*, CoffeeDrinkers)>
<!ELEMENT CoffeeDrinkers (Person*)>

<!ELEMENT Person EMPTY>
<!ATTLIST BusRoute name CDATA #REQUIRED>
<!ATTLIST DieselBus name CDATA #REQUIRED>
<!ATTLIST GasBus name CDATA #REQUIRED>
<!ATTLIST Town name CDATA #REQUIRED>
<!ATTLIST BusStop name CDATA #REQUIRED>
<!ATTLIST Person name CDATA #REQUIRED>

```

The modifications added coffee drinkers that are hanging out at a bus stop but who are not waiting for a bus. The JavaScript program still “works” even after the modifications to the DTD but it gives us the wrong result. The same erroneous behavior we experienced with DemeterJ and AP. Since there is no mechanism to protect against modifications that result in erroneous behavior the only way to ensure proper program functionality is through thorough testing. The idea of Demeter Interfaces is thus applicable and beneficial to XML and XPath queries systems. In a Demeter interface we would use:

```

merge(
    from BusRoute via BusStop
        via -> *,waiting,* to Person,
    from BusRoute via DieselBus to Person)

```

to make sure that only the waiting people are selected at bus stops. For the evolved DTD the interface could be implemented using:

```

var nodes=xmlDoc.selectNodes("./BusStop/Person
    | ./DieselBus//Person")

```

Notice the “/” in the above definition. XPath based tools all suffer from unprotected XPath expressions using // which allows for modifications that render the behavior of the program erroneous. Following the ideas behind Demeter Interfaces and providing implementations for XPath tools, we can remove surprise behavior while at the same time provide higher modularity and better design.

### 3. MODULARITY AND DEMETER INTERFACES

The introduction of Demeter Interfaces to AP development assists in designing, maintaining and understanding adaptive programs. The ideas behind DIs and their usage has revealed several design benefits.

During the design process of adaptive programs developers would first design a minimal class dictionary. Then iteratively both adaptive code and the class dictionary itself are developed with repeated testing to verify the behavior of adaptive code. Modifications to both the class dictionary as well as the adaptive code (the traversal specification and/or the visitor attached) were necessary. As programs become larger in size distinguishing which parts of the class dictionary are involved in the different adaptive methods becomes difficult. Furthermore, modifications to class names in the class dictionary cause changes to traversal strategies and/or visitor methods due to the lack of abstraction over class names. For example, in the development of CONA [10, 11] a Design by Contract (DbC) extension to Java and AspectJ, the class dictionary is the whole Java and AspectJ language. Understanding the dependencies between adaptive code and the class dictionary becomes a laborious and error prone process.

DI provide solutions to both of these problems. The icg provides an abstraction of the concrete class graph while the adaptive methods, traversal strategies and visitor interfaces localize all the information necessary for understanding the dependencies between adaptive code and the rest of the program. The mapping mechanism

removes the tight dependence on naming conventions by providing an automatic renaming mechanism. The usage of DI allows for more modular AP designs.

To support our claim of modularity for DIs we borrow the definition for modular implementations as proposed by Kiczales and Mezini [4]

- it is textually local,
- there is a well-defined interface that describes how it interacts with the rest of the system,
- the interface is an abstraction of the implementation, in that it is possible to make material changes to the implementation without violating the interface.
- an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules, and
- the module can be automatically composed – by a compiler, loader, linker etc. – via various configurations with other modules to produce a complete system.

DIs are textually local with traversal strategies and visitors specifying exactly how adaptive methods interact with the rest of the system. DIs are an abstraction of the implementation both of the class dictionary, through the *icg*, but also through the visitor interfaces that are part of the DI's definition. The extensions made to the DemeterJ system provide automatic mechanisms that both check that modules satisfy their own interfaces as well as the interfaces of other modules. Composition of DI automatically managed by the DemeterJ system and configuration of the composition can be controlled via the *implements* keyword and the mapping specification.

## 4. RELATED WORK

Ovlinger and Wand [9] propose a domain specific language as a means to specify recursive traversals of object structures used with the visitor pattern [3]. The domain specific language further allows for intermediate results from subtraversals allowing for a more functional style visitor definitions. The explicit full definition of the recursive data structure provides an interface between visitors and the underlying data structure. This approach enforces that each object in a traversal is explicitly defined allowing no room for adaptiveness.

Modularity issues in AOSD [4, 1, 12] have received great attention recently. Kiczales and Mezini [4] advocate that in the presence of aspects, a module's interface has to further include pointcuts from aspects that apply to the module in question. These augmented interface definitions, named *aspect-aware interfaces*, can only be determined after the complete configuration of the system's components is known. Aspect-aware interfaces do not provide any extra information hiding capabilities to the base program's modules.

Open Modules [1] extend the traditional notion of a software module to include in its interface pointcut specifications. In this way a module can export, and as such make publicly available, pointcuts within its implementation. This approach gives a balanced control between module and aspect developers in terms of information hiding thus allowing for separate (parallel) evolution of aspects and modules on the agreed upon interface. The interface of a crosscutting concern can affect multiple modules at different join points on each one. Thus an aspect's interface is sprinkled

along module interfaces and not localized making it harder (if not impossible at times) for aspect developers to develop their aspects.

Kevin Sullivan and Bill Griswold and their students [12] advocate an XPI (crosscutting program interface) as a means to achieve separate development and explicit dependencies between implementations of crosscutting concerns and base code. XPIs at present are a design artifact where the agreed upon interface between aspects and base code is explicitly stated using English. Although XPIs assist both during design and development, there is no mechanical checking available to verify the implementation against the agreed upon interface.

Kiczales and Mezini in [5] discuss the benefits of using different programming language mechanisms (procedures, annotations, advice and pointcuts) used to provide separation of concerns at the code level. The resulting guidelines from their analysis sketch the situations where each mechanism will be most effective. The inherent modularity issues associated with each technology are not addressed.

## 5. FUTURE WORK

As immediate future work we would like to complete our implementation of Demeter Interfaces and study extensions to our ideas that would better facilitate AP programming. The set of available predicates that can be used as constraints is still under investigation. Even though with the available set of predicates we could capture all of the constraints that we needed, a bigger set of AP programs needs to be investigated. We are currently extending the language for specifying constraints to include set operators (like subset, proper subset) which can be used on the set of classes that are captured by traversals. Also being able to compose these predicates through logical operators into boolean expressions is a plan for the near future.

Another extension would be the provision of contracts on adaptive methods. Collaborations of adaptive methods exchange data and certain assumptions are being made that are not explicitly captured by the current system. The addition of pre- and post-conditions would assist in both defining and validating these assumptions. The composition of Demeter Interfaces to provide new Demeter Interfaces is also an interesting future research direction. Also the interactions between DI in an AP program still remains an open issue.

A long term goal would be the extension of ideas behind Demeter Interfaces to more general purpose AOP languages. Specifically exposing a graph based model for the control flow of programs upon which *control interfaces* of their expected behavior can be defined. Each program module, *e.g.*, Java class or Java package, will be accompanied by a control interface that will specify an abstraction of the call graph within that module along with constraints. Aspects that are to be attached to this module will have to do so based on the module's interface. A composition of two modules will generate a composition of their corresponding control interfaces. Aspect developers and class developers will first agree on the control interface of their modules and the assumptions that it must uphold. The implementation of the module and the aspects around it will then be checked against the agreed upon control interface. Similar ideas for interfaces between aspects and base code have been recently discussed [1, 12, 5].

## 6. CONCLUSION

We introduce Demeter Interfaces as an extension to Adaptive Programming. Demeter interfaces encapsulate all the information and dependencies between the adaptive code and the underlying data structure. Through the definition of an interface class graph

and a set of graph constraints Demeter Interfaces impose restrictions on any concrete data structure to which adaptive code will be attached. These restrictions are enforced at compile time disallowing modifications to the underlying data structure that would otherwise provide incorrect program results. With Demeter interfaces in place we have shown how modularity as well as understandability of adaptive programs increases dramatically leading to better program design and promotes parallel development.

## 7. REFERENCES

- [1] Jonathan Aldrich. Open Modules: modular reasoning about advice. In *European Conference on Object-Oriented Programming*, 2005.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.
- [4] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming*, 2005.
- [5] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X.
- [6] Mira Mezini and Karl J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, number 10, pages 97–116, Vancouver, October 1998. ACM.
- [7] Doug Orleans and Karl J. Lieberherr. DemeterJ. Technical report, Northeastern University, 1996-2001. <http://www.ccs.neu.edu/research/demeter/DemeterJava/>.
- [8] Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 70–81, New York, NY, USA, 1999. ACM Press.
- [9] Therapon Skotiniotis and David Lorenz. Conaj: Generating contracts as aspects. Technical Report NU-CCIS-04-03, College of Computer and Information Science, Northeastern University, 2004.
- [10] Therapon Skotiniotis and David H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 196–197, New York, NY, USA, 2004. ACM Press.
- [11] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nisit Tewan, and Hriday Rajan. On the criteria to be used in decomposing systems into aspects. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2005.