Practical Challenges in Object-Oriented Dependence Analysis

Neil Walkinshaw

Department of Computer Science
University of Leicester

Slicing COW, January, 2011

University of Leicester

Introduction

Context

- OO paradigm is the de-facto choice for software development
- Subject to several factors that hamper dependence analysis

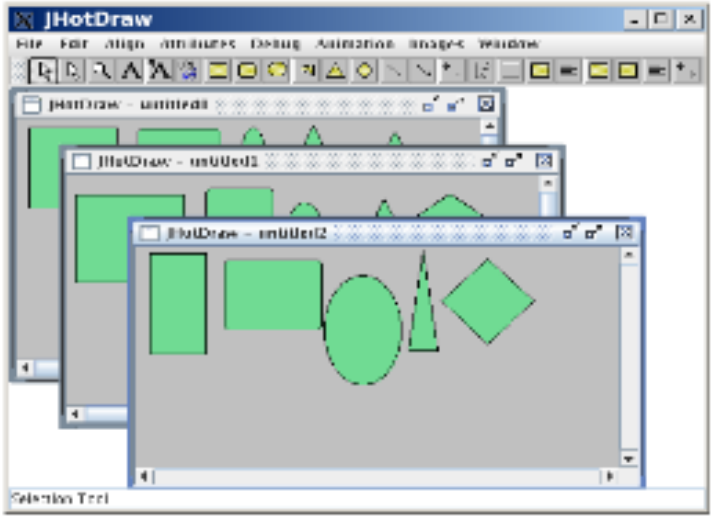
Aim of the talk

- To show problems that arise with static analysis of OO code
- From the perspective of my thesis work
  - Develop a source code reading tool
  - Purpose: To navigate the source code related to a given system-level feature

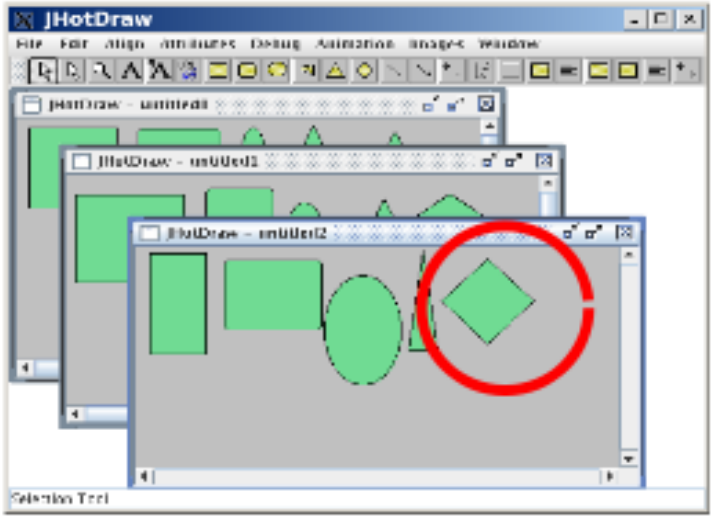
To bear in mind...

- Thesis work carried out 2002-2005
  - Some opinions could be out of date
- Have tried to link relevant slides to discussion on functional slicing
  - Relevant slides marked

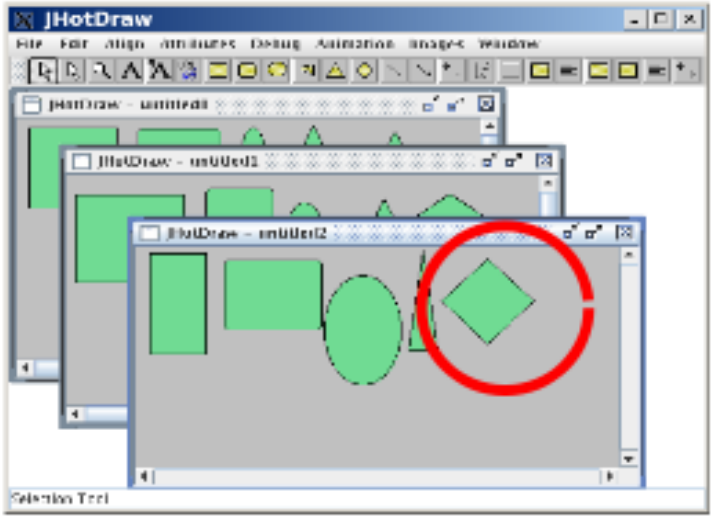
The Challenge of OO Feature Identification



The Challenge of OO Feature Identification



The Challenge of OO Feature Identification

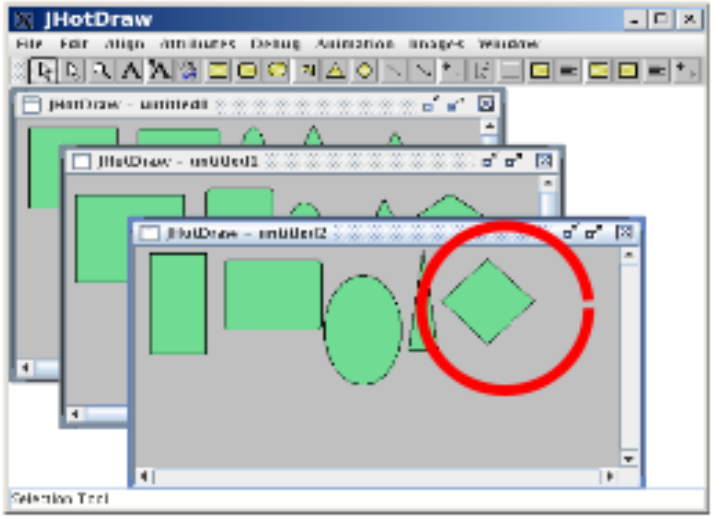


370 classes

1752 methods

3203 calls

The Challenge of OO Feature Identification



370 classes

1752 methods

3203 calls

How to identify and navigate through the code corresponding to the diamond tool?

Thesis idea

Use static code analysis, with limited input from developer

- Combine slicing, call graph analysis and developer input
- Should identify relevant methods and method calls
- Integrate into a code-reading interface

Thesis idea

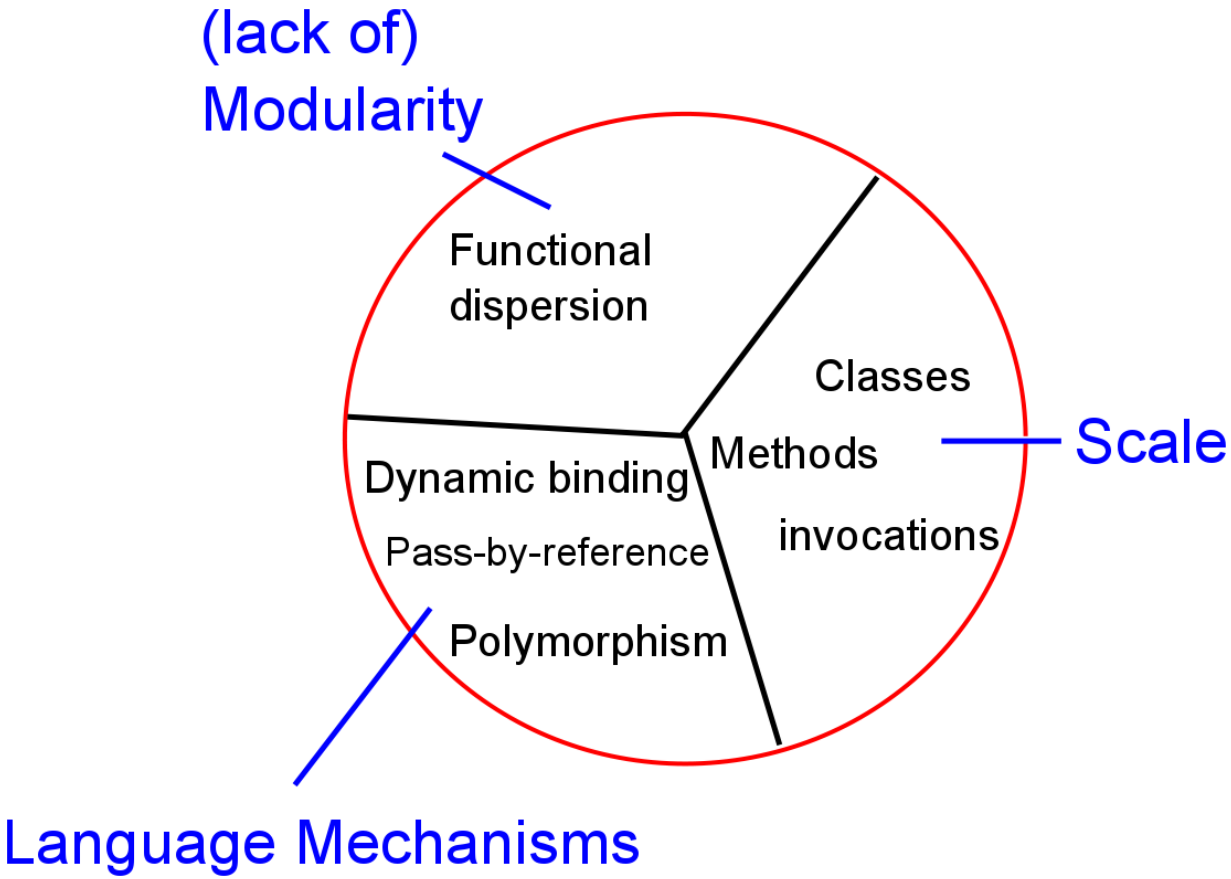
Use static code analysis, with limited input from developer

- Combine slicing, call graph analysis and developer input
- Should identify relevant methods and method calls
- Integrate into a code-reading interface

Outcome

- Possible to produce an accurate result
  - ... but only with large amount of help from the developer
- Problem massively exacerbated as scale increased

OO Static Analysis Challenges



Problems Posed by OO Mechanisms

Polymorphism and Dynamic Binding

- Type of an object is defined by the interface it provides
  - Implementations for methods declared in hierarchy
  - Methods bound to their implementations at runtime

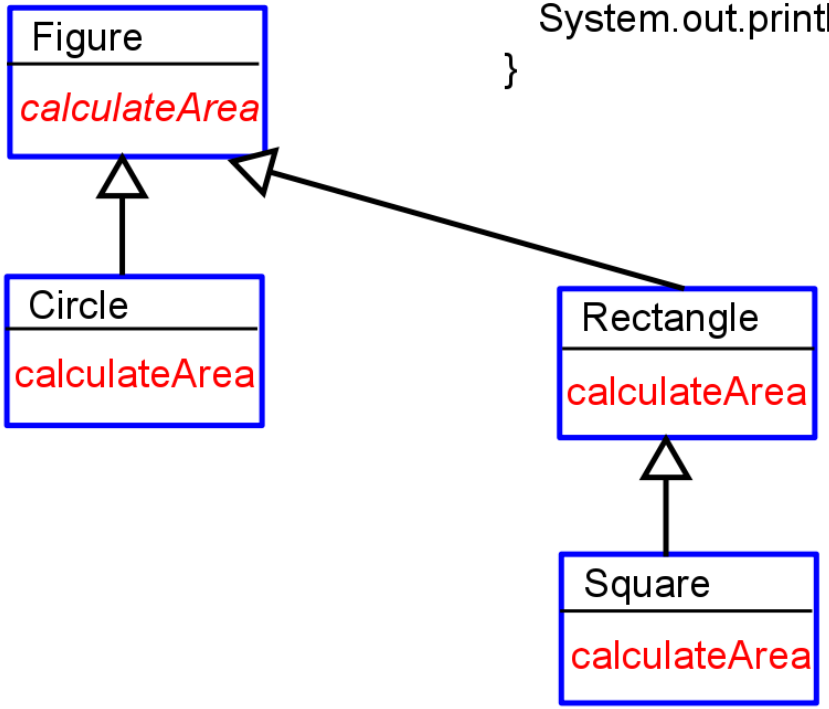
```
public void printArea(Figure f){
   double area = f.calculateArea();
   System.out.println(area);
}
```

Problems Posed by OO Mechanisms

Polymorphism and Dynamic Binding

- Type of an object is defined by the interface it provides
  - Implementations for methods declared in hierarchy
  - Methods bound to their implementations at runtime

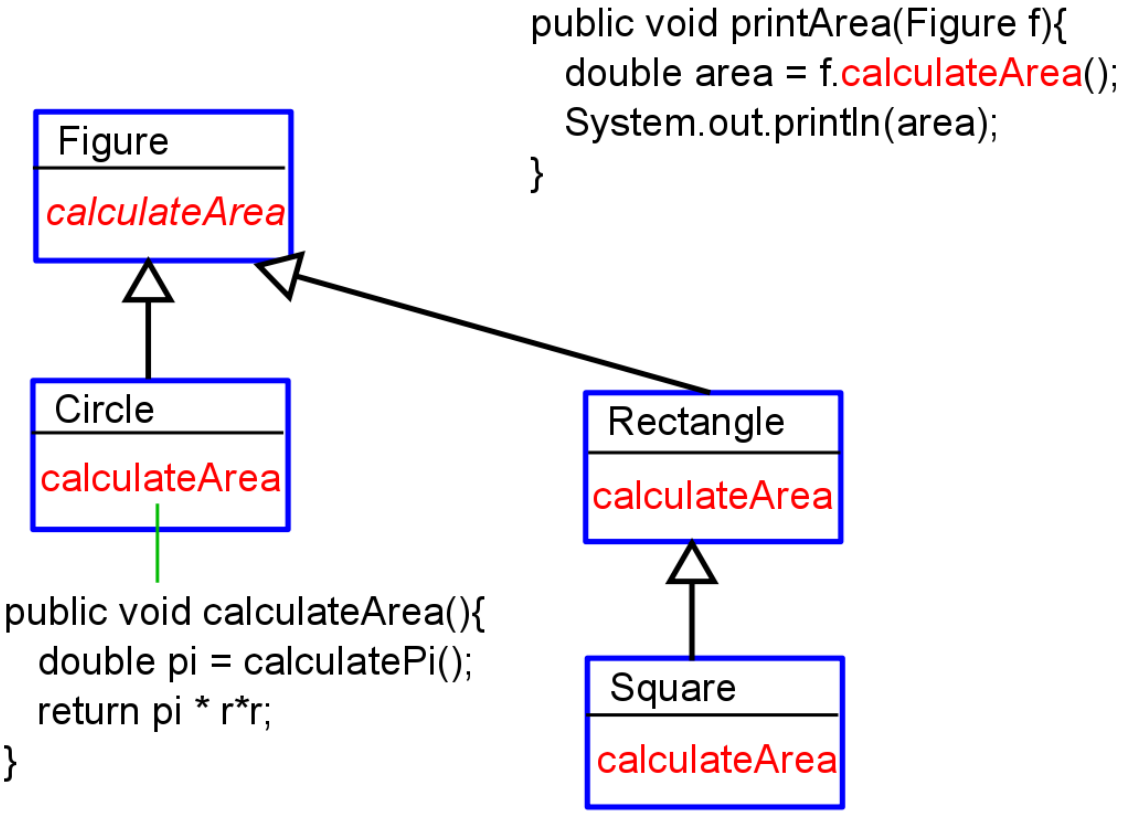
```
public void printArea(Figure f){
   double area = f.calculateArea();
   System.out.println(area);
}
```



Problems Posed by OO Mechanisms

Polymorphism and Dynamic Binding

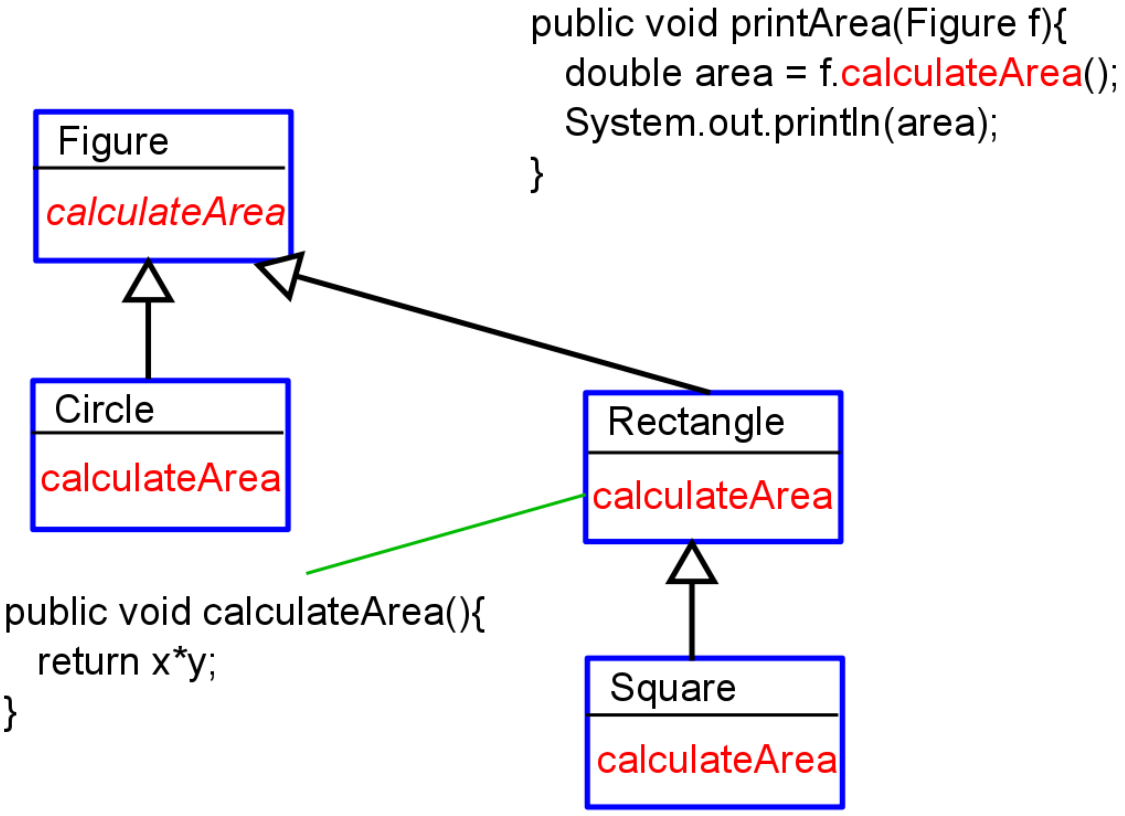
- Type of an object is defined by the interface it provides
  - Implementations for methods declared in hierarchy
  - Methods bound to their implementations at runtime



Problems Posed by OO Mechanisms

Polymorphism and Dynamic Binding

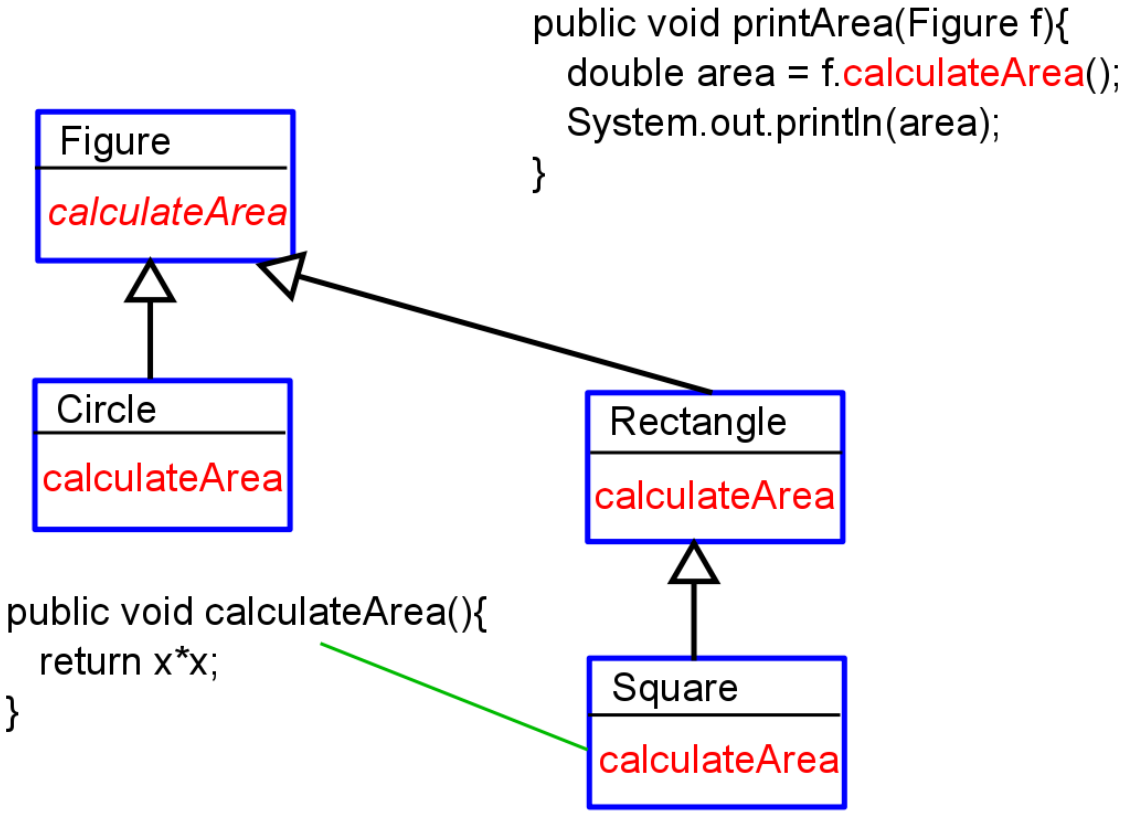
- Type of an object is defined by the interface it provides
  - Implementations for methods declared in hierarchy
  - Methods bound to their implementations at runtime



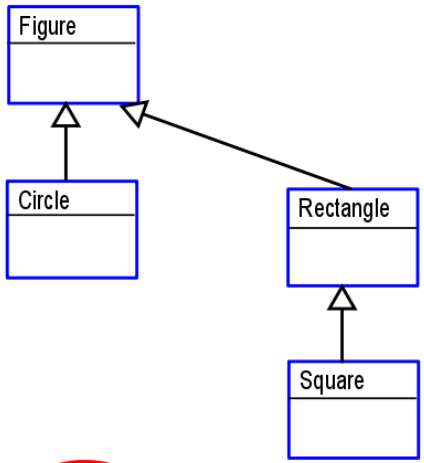
Problems Posed by OO Mechanisms

Polymorphism and Dynamic Binding

- Type of an object is defined by the interface it provides
  - Implementations for methods declared in hierarchy
  - Methods bound to their implementations at runtime



Problems Posed by OO Mechanisms



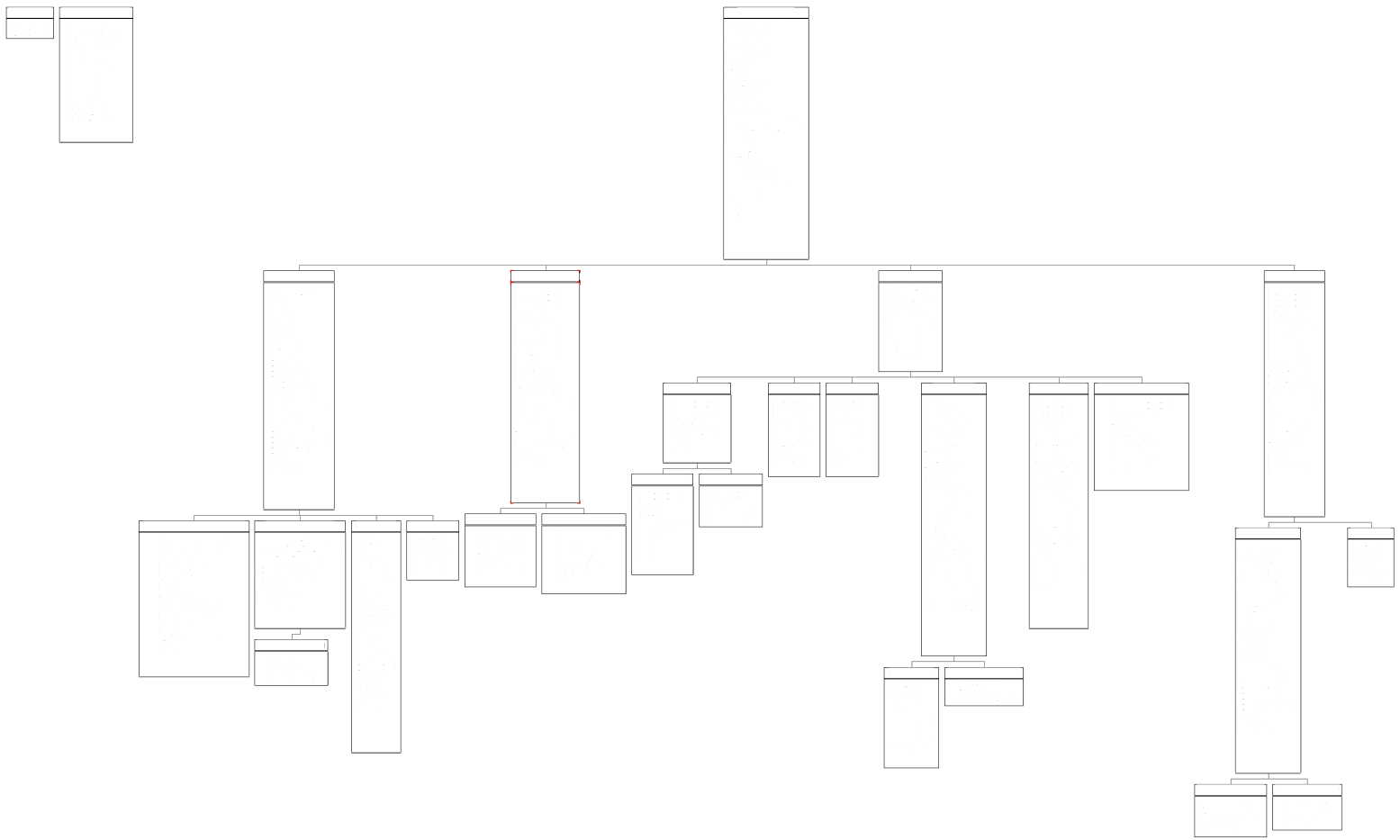
```
public void printArea(Figure f){
   double area = f.calculateArea();
   System.out.println(area);
   plot(f);
}

public void plot(Circle c){
    …
}

public void plot(Rectangle r){
    …
}

public void plot(Square s){
    …
}
```

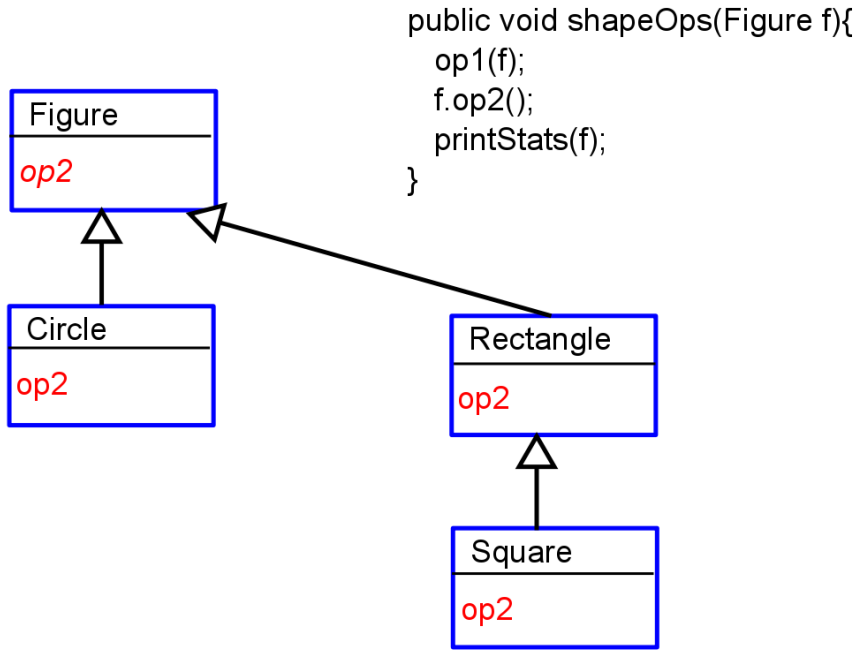
Problems Posed by OO Mechanisms



Problems Posed by OO Mechanisms

Data dependence computation relies on inter-procedural analysis

- A **def** or a **use** of a variable can depend on the called method
  - Does call to an object method mutate or access it?
  - If passed as an argument, is it accessed or mutated?
- Demands inter-procedural analysis
  - Erroneous pointers can result in def-use errors
  - Static analysis forced to include library methods



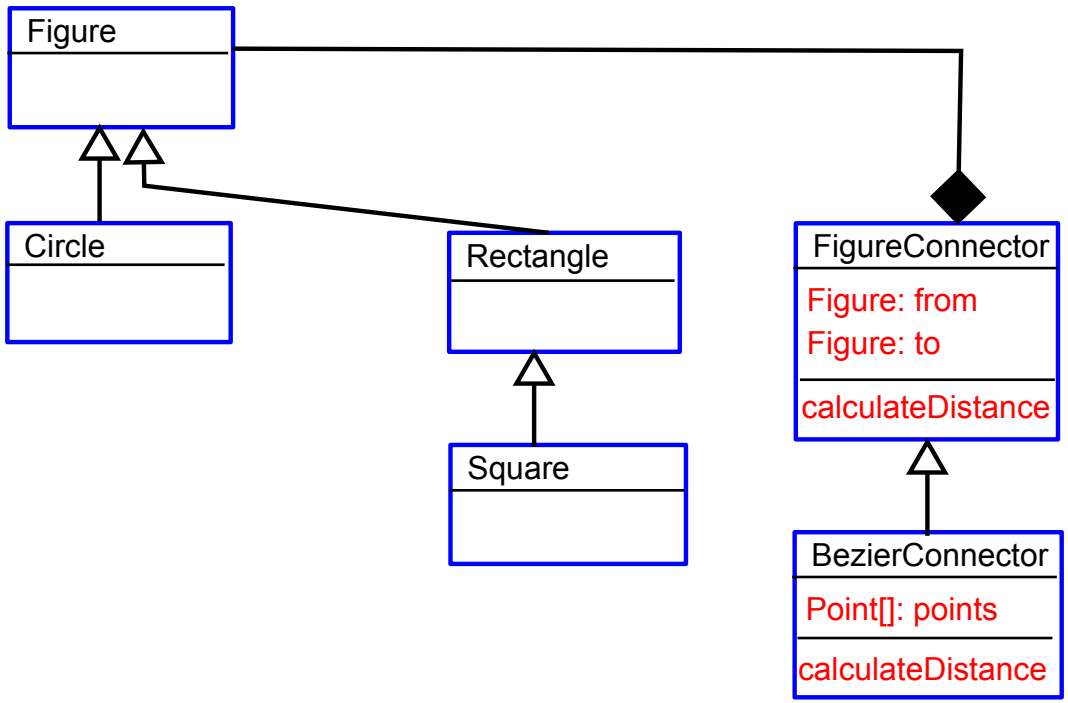
Problems Posed by OO Mechanisms

Object associations

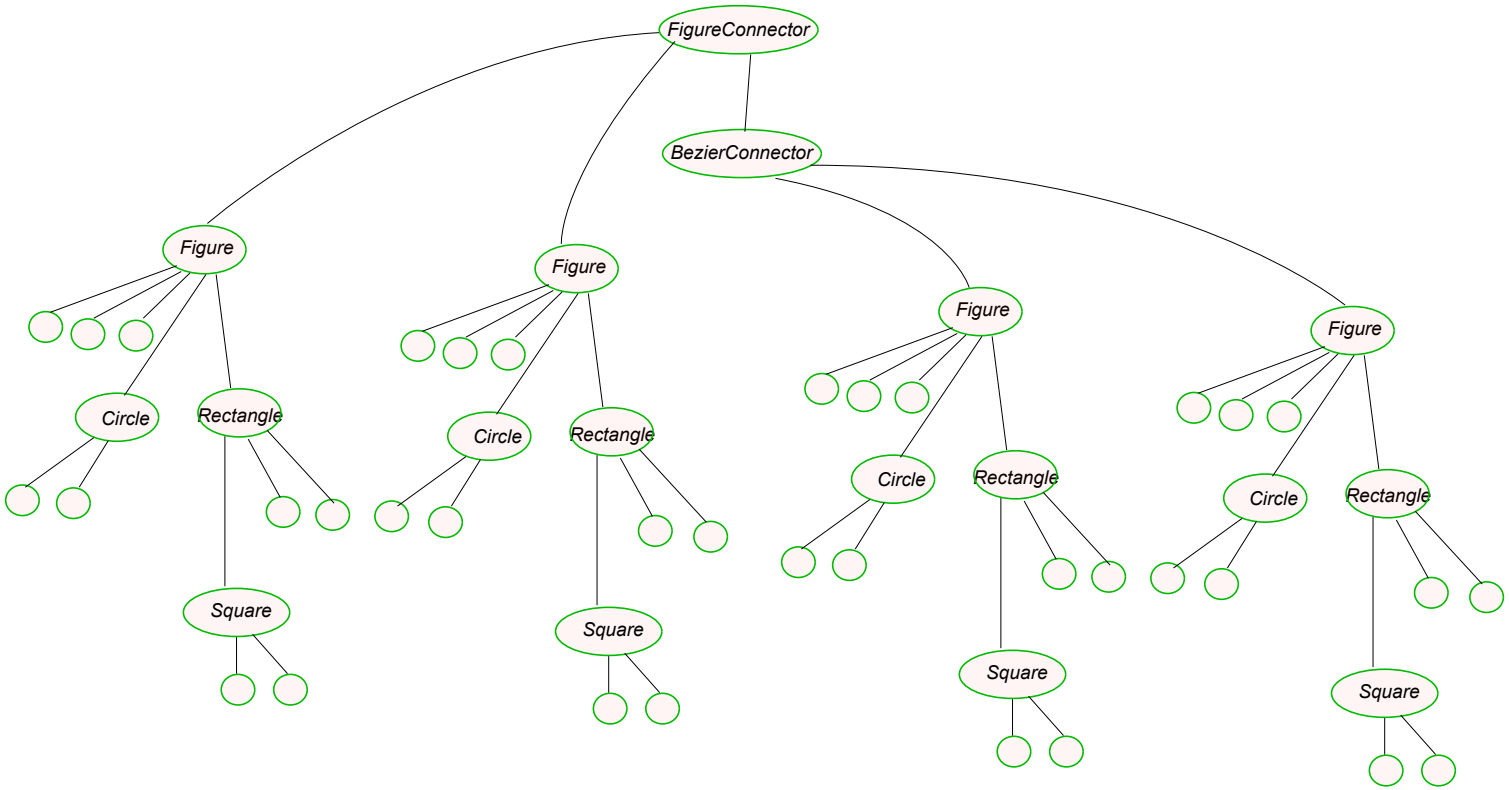
- Objects do not occur in isolated hierarchies
  - Objects can contain other objects belonging to different hierarchies
- The possible types of the associated objects (and their objects) have to be accounted for
- Dependence analysis needs to account for every possible combination of types
  - Parameter trees

Problems Posed by OO Mechanisms

```
printDistance(FigureConnector fc){
    double distance = fc.calculateDistance()
    System.out.println(distance);
}
```



Problems Posed by OO Mechanisms



Delocalised Design

Dependencies are heavily delocalised

- Classes of objects represent units that are conceptually aligned with problem domain
  - Dependencies tend to span the system
  - Some evidence that OO dependencies obey principles of small-world networks
    - "Software systems as complex networks...", C. Myers, 2003

Delocalised Design

Dependencies are heavily delocalised

- Classes of objects represent units that are conceptually aligned with problem domain
  - Dependencies tend to span the system
  - Some evidence that OO dependencies obey principles of small-world networks
    - "Software systems as complex networks...", C. Myers, 2003

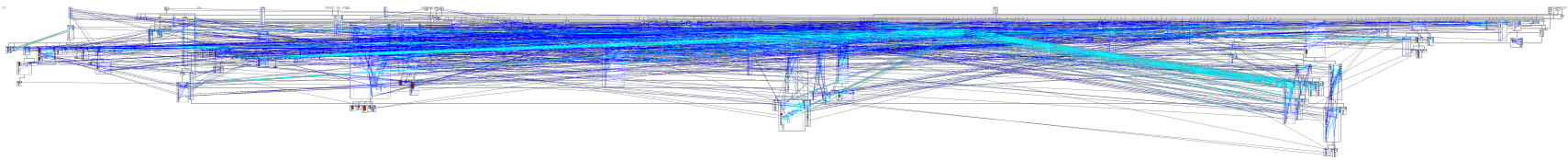
- *"Distribute system intelligence horizontally as uniformly as possible, that is, the top level classes in a design should share the work uniformly."* [Riel, OO design heuristics, Addison Wesley 2002]

Delocalised Design

Dependencies are heavily delocalised

- Classes of objects represent units that are conceptually aligned with problem domain
  - Dependencies tend to span the system
  - Some evidence that OO dependencies obey principles of small-world networks
    - "Software systems as complex networks...", C. Myers, 2003

- *"Distribute system intelligence horizontally as uniformly as possible, that is, the top level classes in a design should share the work uniformly."* [Riel, OO design heuristics, Addison Wesley 2002]



Scale

Good OO design fosters fragmentation 

- Large classes and long methods are "code smells"
- Small classes and methods facilitate reuse and program understanding
  - Remedy: Break the system up into smaller classes, with smaller methods

Scale

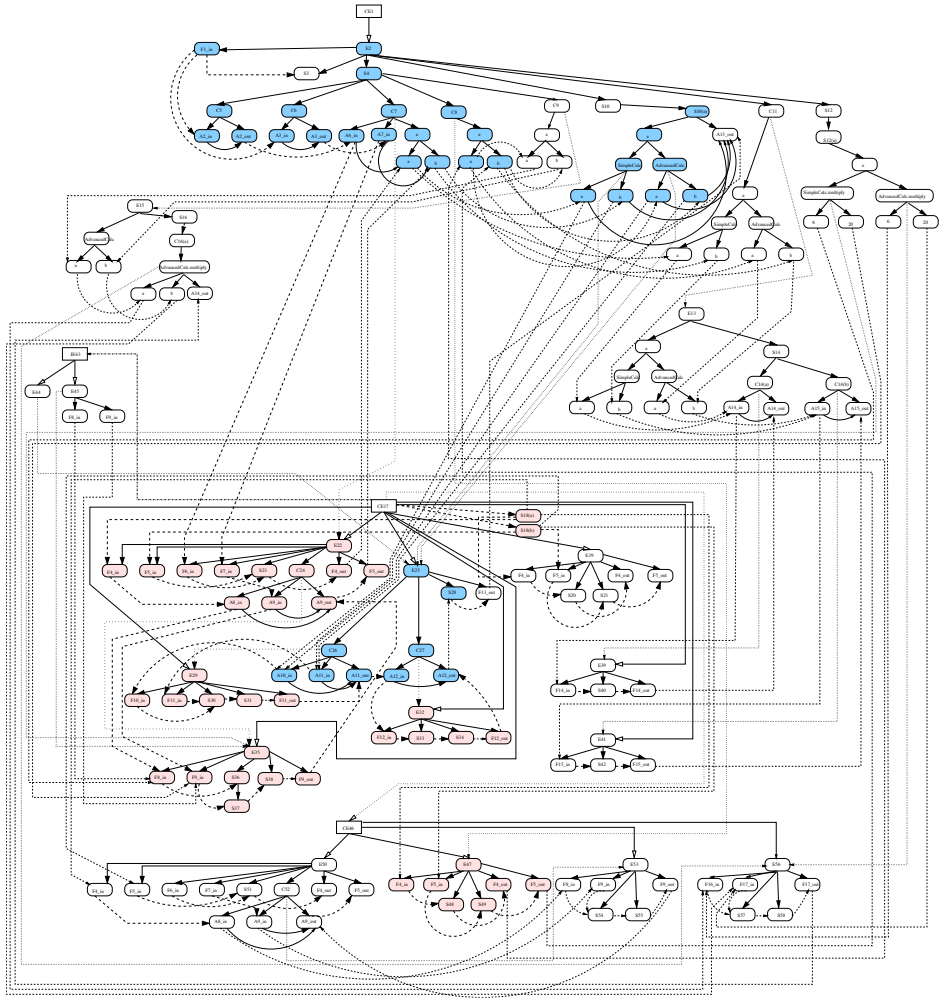
Good OO design fosters fragmentation

- Large classes and long methods are "code smells"
- Small classes and methods facilitate reuse and program understanding
  - Remedy: Break the system up into smaller classes, with smaller methods

Results in a problem of scale

- Dependence graph size for procedural programs is more or less linear
- This is certainly not the case for object-oriented programs:
  - More classes and methods with their respective parameter vertices
  - Many polymorphic calls (with parameter edges)

Scale



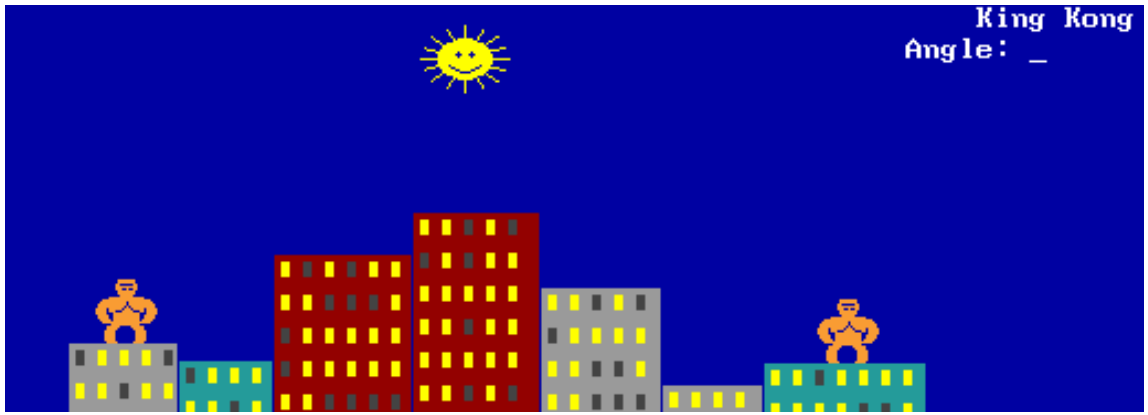
Summary

- Conspiracy of three problems:
  1. Accurate prediction of runtime dependencies from code is impossible
  2. Dependencies tend to cut across the system
  3. Granular decomposition causes scalability problems
- Each ill-computed dependency will significantly amplify problems 2 and 3
- The better designed a system is, the worse these problems become

Summary

> *"The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."*

– Joe Armstrong



Conclusions

- Accuracy of dependence analysis intricately tied to accuracy of underlying analyses
  - Results tend to be grossly inaccurate
- Cannot be ignored when interpreting OO dependence analysis results
  - Slices can easily become misleading
  - Implications for comprehension tasks
  - Implications for slice-based metrics?
- Demands new OO dependence analysis techniques
  - Or at least new ways to interpret their results

Several of potential solutions

- Associate confidence measure with each dependence?
- Incorporate dynamic analysis - trace / profiling information
  - Can use test-set executions

Several of potential solutions

- Associate confidence measure with each dependence?
- Incorporate dynamic analysis - trace / profiling information
  - Can use test-set executions
- Incorporate infrastructures of code-based model-checking platforms
  - Exploit symbolic reasoning capacities of model-checking platforms such as JPF, Bandera etc.
  - These enable the incorporation of invariants, behavioural models etc. into the analysis
  - Use their ability to provide abstract models of irrelevant classes