CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2018
Doc 17 Mediator, Flyweight, Facade, Demeter, Active Object
Nov 19, 2019

References

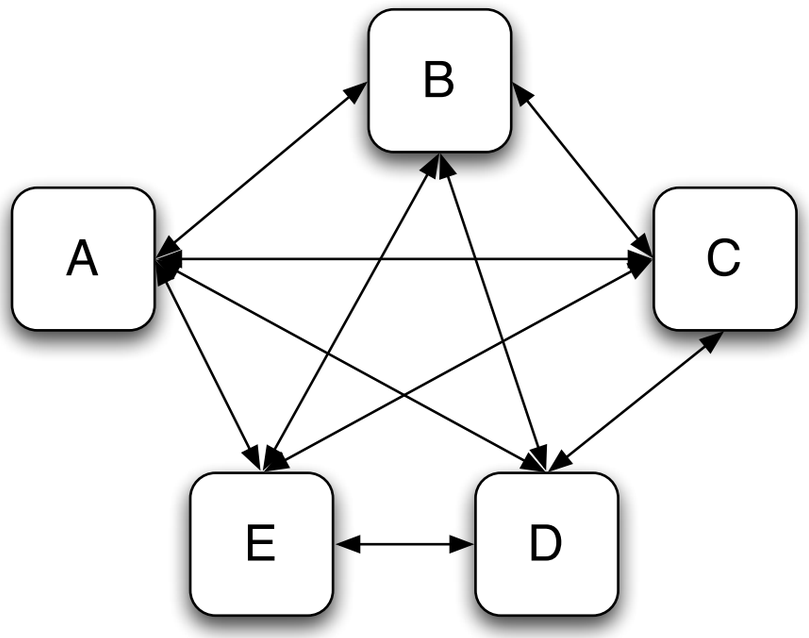
Metadata and Active Object Models, Foote & Yoder, http://hillside.net/plop/plop98/final_submissions/P59.pdf

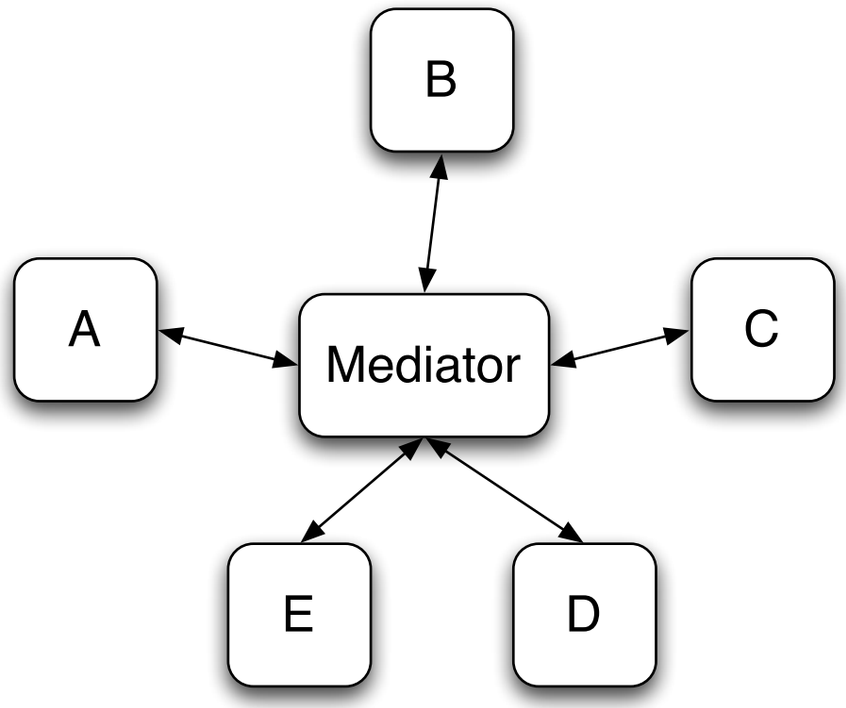
The User-Defined Product Framework, Johnson & Oakes, https://www.researchgate.net/publication/2640344_The_User-Defined_Product_Framework

Mediator

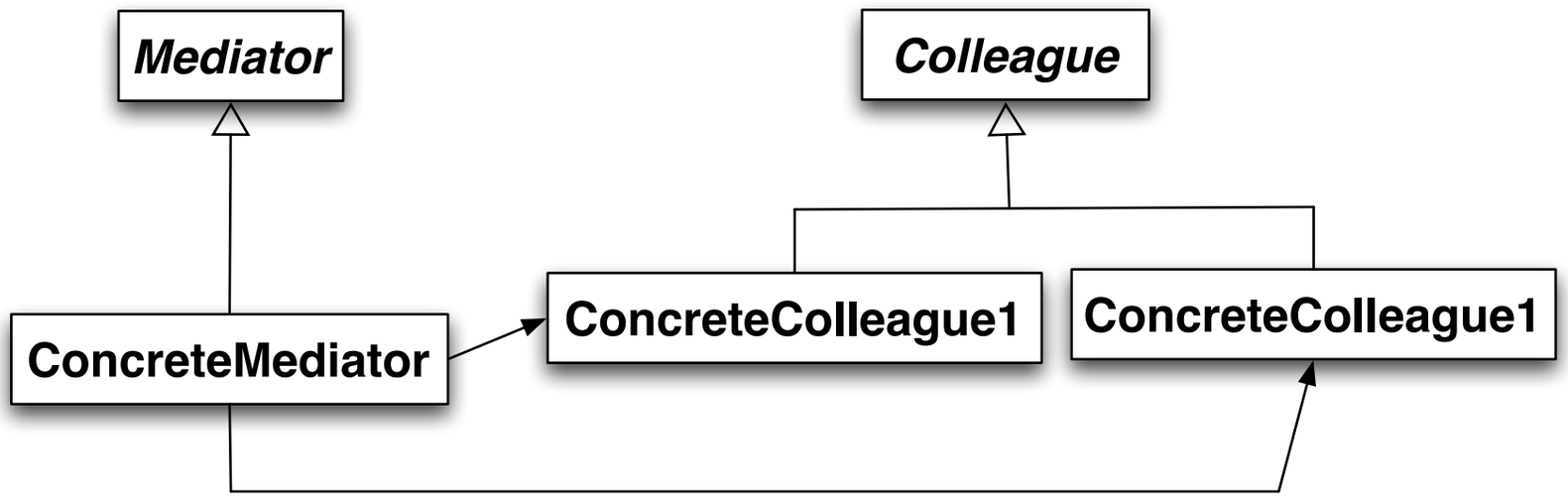
Mediator

A mediator controls and coordinates the interactions of a group of objects





Structure


Mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague1

Participants

Mediator

Defines an interface for communicating with Colleague objects

ConcreteMediator

Implements cooperative behavior by coordinating Colleague objects

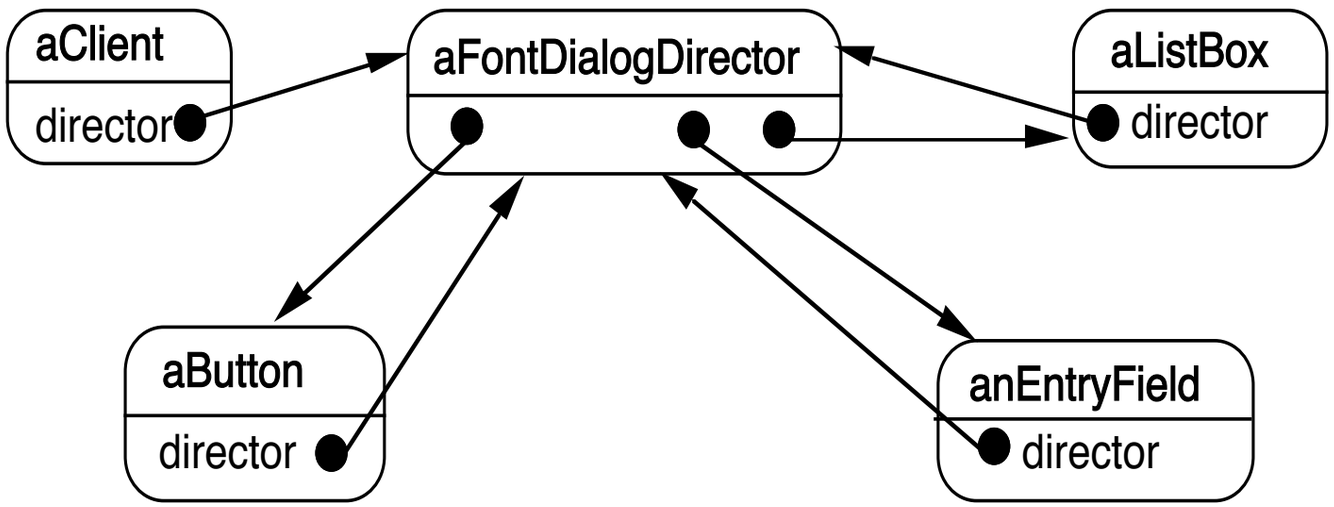
Knows and maintains its colleagues

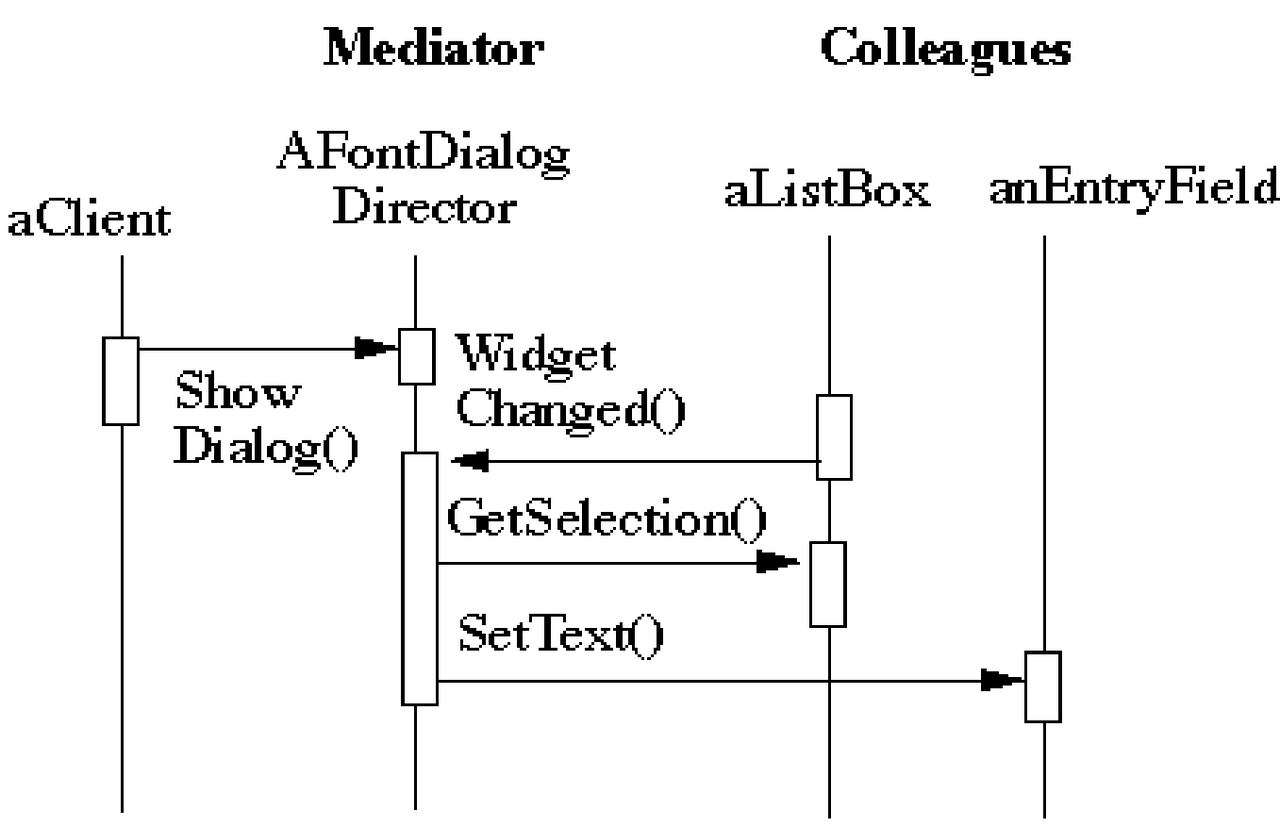
Colleague classes

Each Colleague class knows its Mediator object

Each colleague communicates with its mediator whenever it would have otherwise communicated with another colleague

Motivating Example - Dialog Boxes





How does this differ from a God Class?

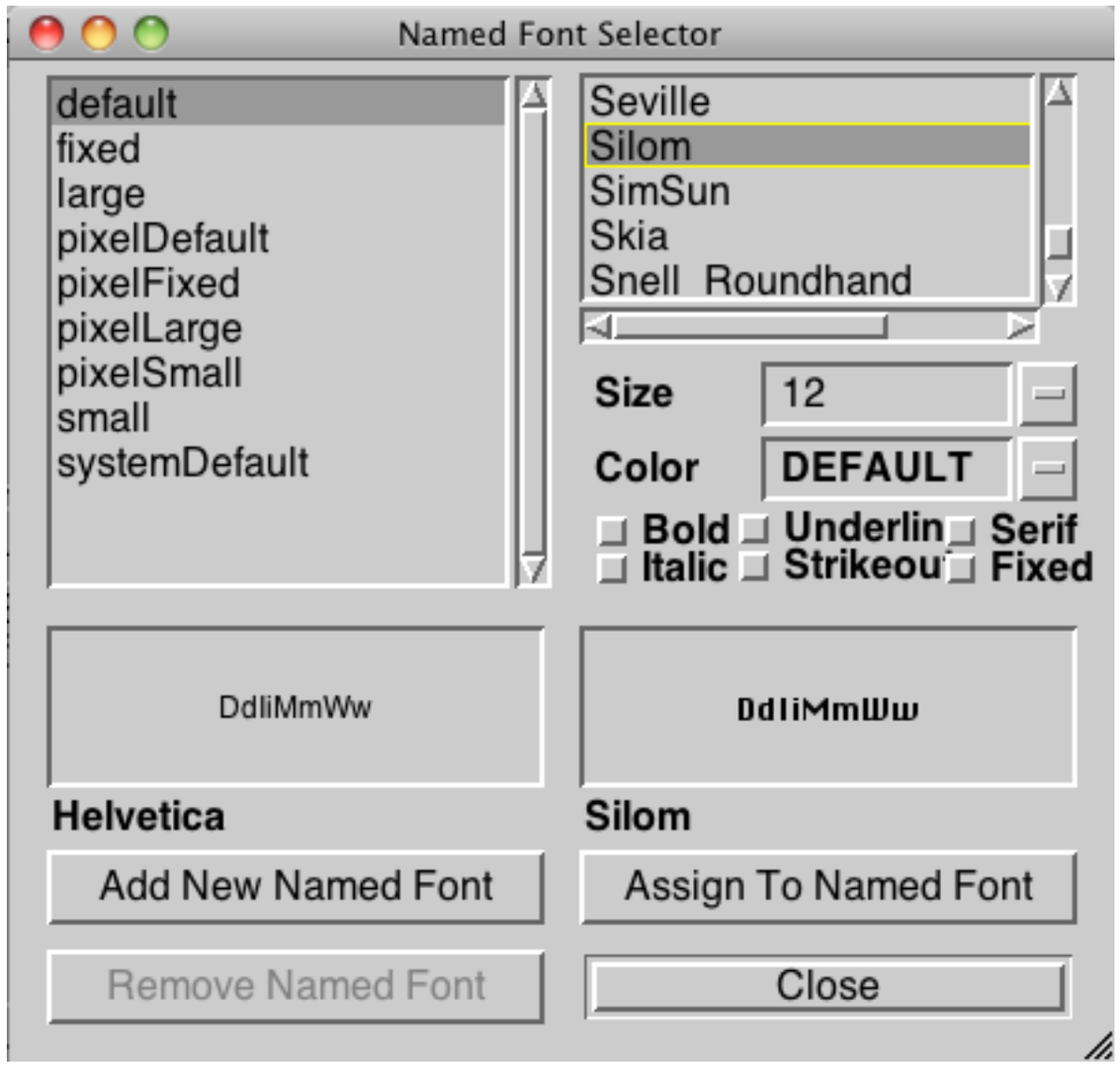
When to use the Mediator Pattern

When a set of objects communicate in a well-defined but complex ways

When reusing an object is difficult because it refers to and communicates with many other objects

When a behavior that's distributed between several classes should be customizable without a lot of subclassing

Classic Mediator Example

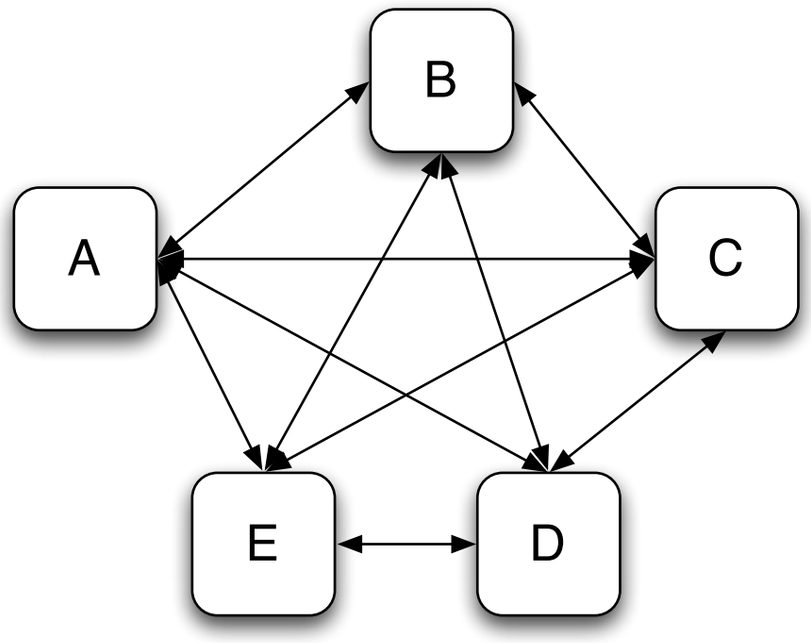


Simpler Example

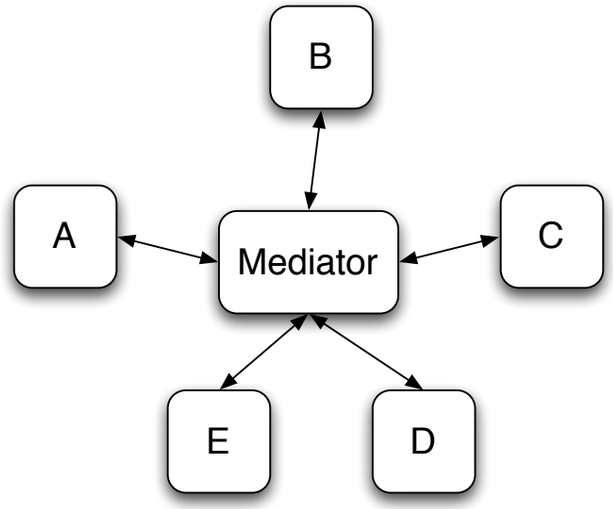
Non Mediator Solution

```
class OKButton extends Button {
   TextField password;
   TextField username;
   Database userData;
   Model application;

   protected void processEvent(AWTEvent e) {
      if (!e.isButtonPressed()) return;
      e.consume();
      if (password.getText() = "") {
   notifyUser("Must enter password");
   return;
}
if (username.getText() = "") {
   notifyUser("Must enter user name");
   return;
}
if (!userData.validUser(password.getText(), username.getTest()))
   notifyUser("Invalid username & password");
   return;
}
```



Mediator Solution




```
class LoginDialog extends Panel {
   TextField password;
   TextField username;
   Database userData;
   Button ok, cancel;

   protected void actionPerformed(ActionEvent e)  {
      if (!e.isButtonPressed() or e.getSource() != ok) return;
      if (password.getText() = "") {
   notifyUser("Must enter password");
   return;
}
if (username.getText() = "") {
   notifyUser("Must enter user name");
   return;
}
if (!userData.validUser(password.getText(), username.getTest()))
   notifyUser("Invalid username & password");
   return;
}
```

What is Different?

Non Mediator Example

- Special Button class
- OK button coupled to text fields

Mediator Example

- No specialButton class
- LoginDialog coupled to text fields

Logic moved from button class to LoginDialog

ReactiveX

In some cases ReactiveX reduces mediator to setting up streams

Facade

Size

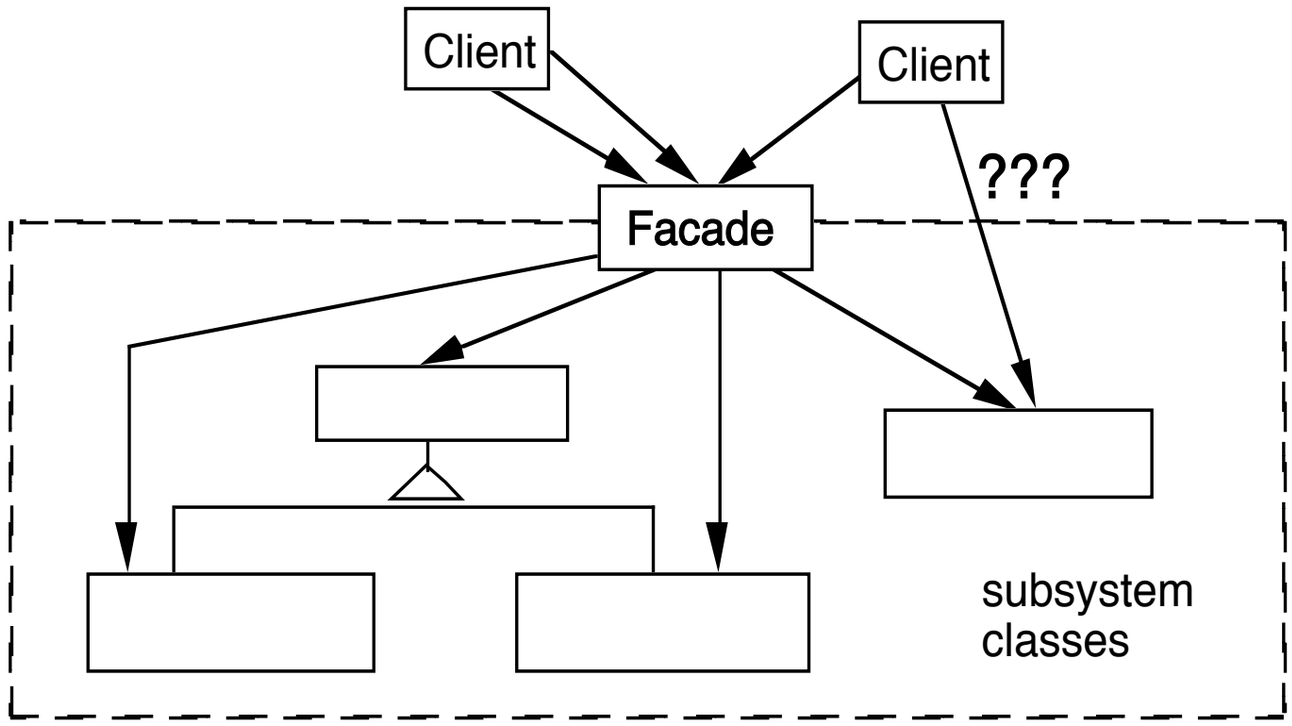
Item	Source Lines of Code (Millions)
F-22 Raptor US jet fighter	1.7
Boeing 787	6.5
Chevy Volt - Embedded Code	10
S-class Mercedes-Benz radio & navigation system	20
Mac OS 10.4	86
New automobile	~100
Debian 5.0	342
Tesla	Linux + ?

Design Patterns text contains under 8,000 lines

The Facade Pattern

Create a class that is the interface to the subsystem

Clients interface with the Facade class to deal with the subsystem



Consequences of Facade Pattern

It hides the implementation of the subsystem from clients

It promotes weak coupling between the subsystems and its clients

It does not prevent clients from using subsystem classes directly, should it?

Facade does not add new functionality to the subsystem

Public versus Private Subsystem classes

Some classes of a subsystem are

- public
- facade
- private

Compiler Example

The VisualWorks Smalltalk compiler system has 75 classes

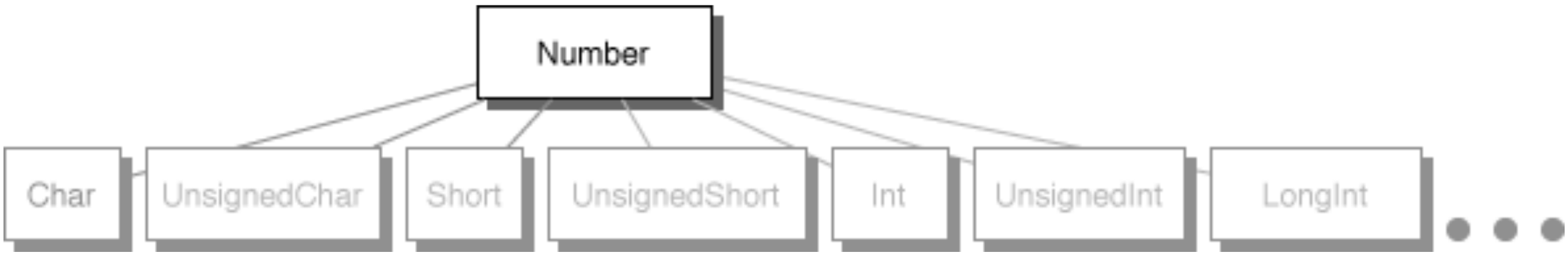
Programmers only use Compiler, which uses the other classes

Compiler evaluate: '100 factorial'

```
| method compiler |
method := 'reset
    "Resets the counter to zero"
    count := 0.'.

compiler := Compiler new.
compiler
    parse:method
    in: Counter
    notifying: nil
```

Objective-C Class Clusters & Facade



Law of Demeter

Law of Demeter

A method M of object O can only call methods on the following objects

O
Arguments of M
Objects created within M
O's direct component objects
A global variable

Law of Demeter

Use only one dot

a.method();

a.b.method();

a.methodB().methodC();

foo = a.methodB();
foo.methodC();

What about Builder Example?

```
Notification note = new Notification.Builder(mContext)
        .setContentTitle("New mail from " + sender.toString())
        .setContentText(subject)
        .setSmallIcon(R.drawable.new_mail)
        .setLargeIcon(aBitmap)
        .build();
```

What about Builder Example?

Each method returns the builder

```
Notification.Builder mailNotifcation= new Notification.Builder(mContext);
mailNotifcation.setContentTitle("New mail from " + sender.toString());
mailNotifcation.setContentText(subject);
mailNotifcation.setSmallIcon(R.drawable.new_mail);
mailNotifcation.setLargeIcon(aBitmap);
Notification note = mailNotifcation.build();
```

Hinges

Business Rules

Some businesses frequently change rules/deals

Buy two X and get third X for 1/2 price

20 cent coffee day

Don't have time to rewrite code

Need to move business logic into data

Metadata and Active Object Models

Metaprogramming

"Writing of computer programs that write or manipulate other programs (or themselves) as their data"

Wikipedia

Forces in Software Evolution

Make programs as general as possible

Push config decisions
- into the data
- To users
- Defer until runtime

Property Pattern

Property

Attributes
Annotations
Dynamic Slots
Property List

How do you allow individual objects to augment their state at runtime

Therefore, provide runtime mechanisms for accessing, altering, adding, and removing properties or attributes at runtime

What is a Property?

Key (Indicator) - name of the property

Value - the value of the property

Descriptor - information about property
  display name, type, constraints
  default value, accesor functions, etc

  Indicates how to downcast
  Used by tools

Java Example (Fake)

```
class Example {
  HashMap<String,Object> properties = new Hashmap<String, Object>();

  public void setProperty(String name, Object value) {
    properties.put(name, value);
  }

public Object getProperty(String name) {
  return properties.get(name);
}

public boolean hasProperty(String name) {
  return properties.containsKey(name);
}
```

Some Property methods

void addProperty(Indicator name, Descriptor aboutProperty, Object value);
void removeProperty(Indicator name);
boolean hasProperty(Indicator name);
void setProperty(Indicator name, Object value);
Object getProperty(Indicator name);

Decriptor getDescriptor(Indicator name);
Descriptor[] getDescriptors();
Object[] propertyList();

Java Properties Class

```
Properties defaults = new Properties();
defaults.put("a", "one");
defaults.put("b", 'two");

Properties test = new Properties(defaults);
test.put("c", "three");
test.put("a", "override a default");

test.get("a");
test.get("b");
test.get("d");
```

Consequences

You avoid a proliferation of subclasses

Fields may be added to individual instances

Fields may be added and removed at runtime

You may iterate across the fields

Metainformation is available to facilitate editing and debugging

Properties can graduate to first-class fields as an application evolves.

Consequences

Syntax is more cumbersome in the absence of reflective support

Property access code is more complex that that for real fields

Reflective mechanisms, where they are available, can be slower

Idiomatic implementations, when reflective support is not available, are also slow

Access to heterogeneous collections can be expensive

A field must be added to all objects, while only a few ever use it

The User-Defined Product Framework

The User-Defined Product Framework

Let users

Construct a complex business object from existing components

Define a new kind of component without programming

Insurance managers can invent a new policy rider

Framework developed at ITT Hartford

Used to represent insurance policies

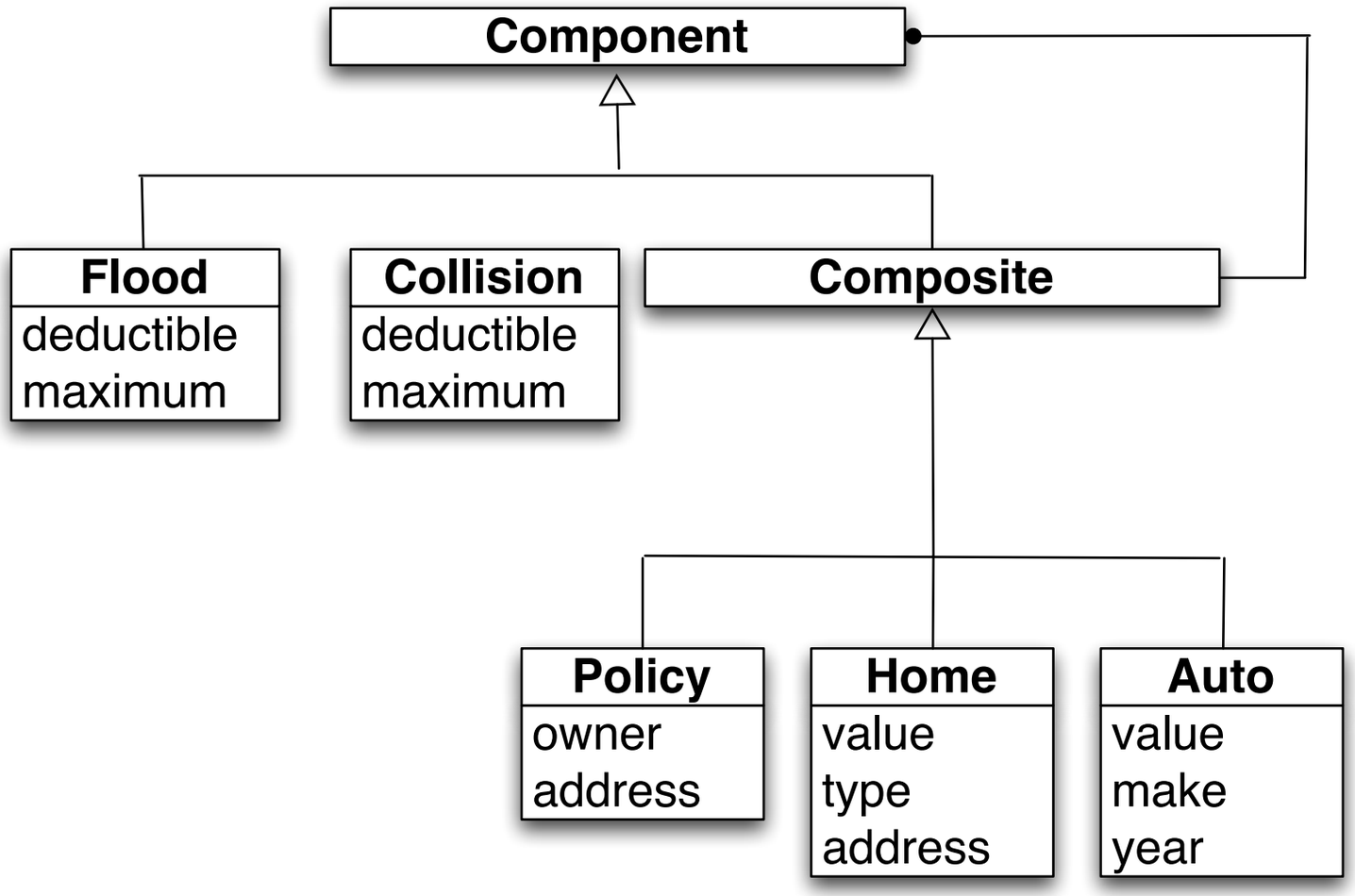
Problem

Which is the best way to combine features, multiple inheritance or composition?

Need 10,000 classes to get all the combinations needed

Use object composition to combine features instead of multiple inheritance.

Solution - Composition



Policy
Auto
Collision
Auto
Home
Flood

Problem

Design is still complex and hard to use

a huge number of Component classes

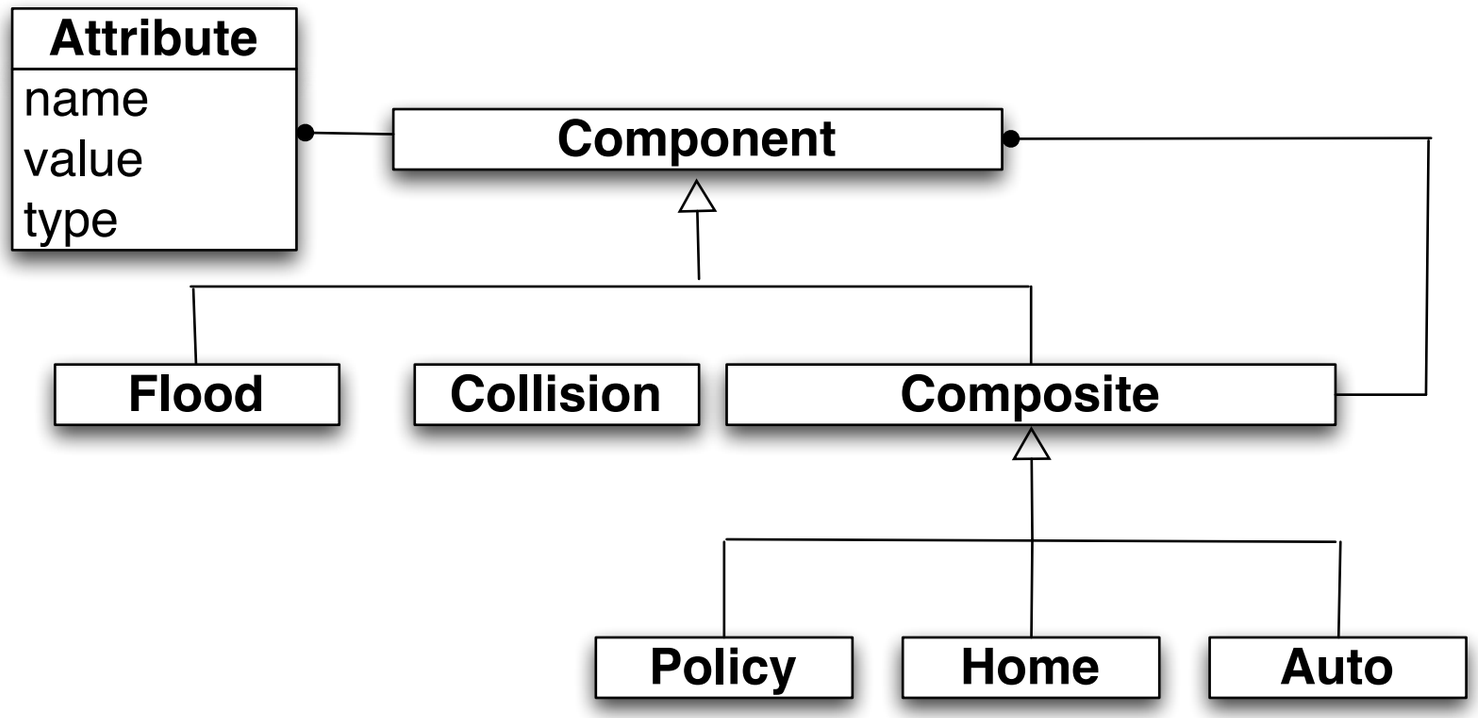
adding a feature means making a new one

Component has too many subclasses.

How can we keep from having to subclass Component?

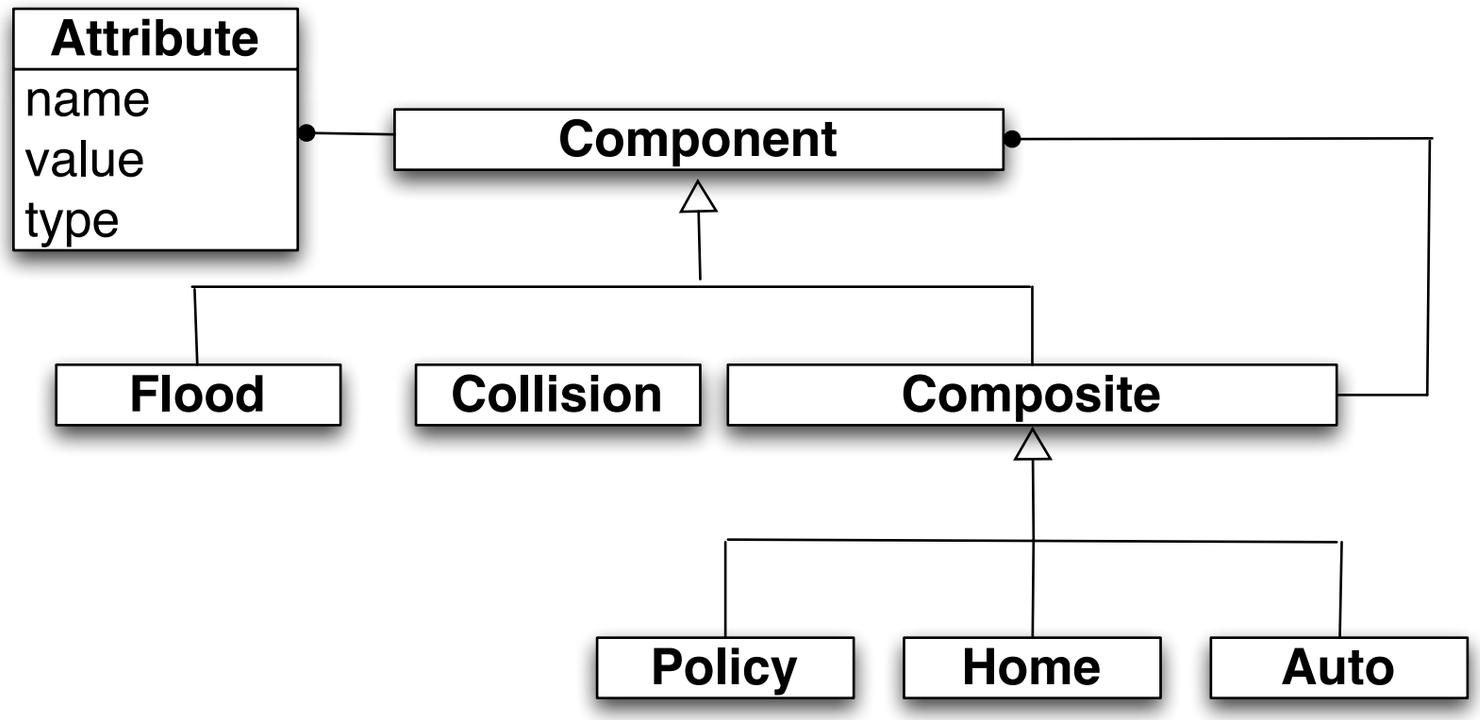
Solution - Properties (Variable State)

Eliminate the need to subclass to add instance variables by storing attributes in a dictionary instead of directly in an instance variable.



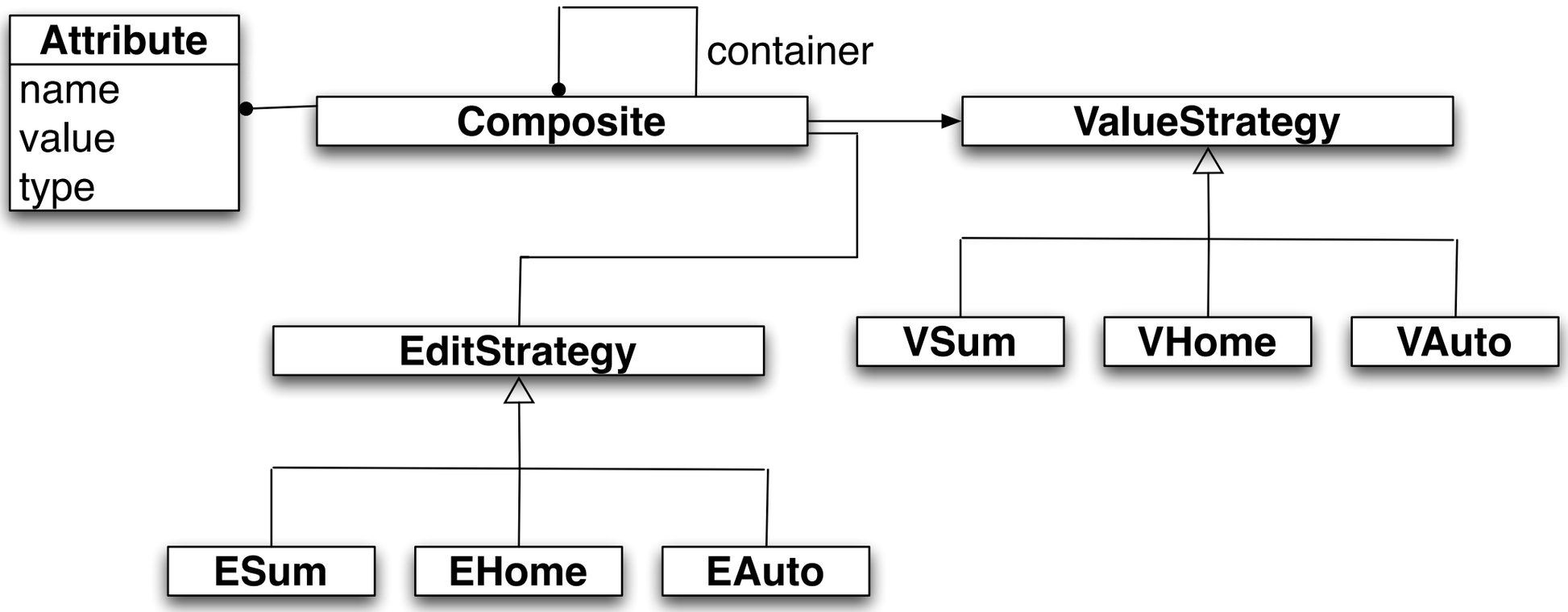
Problem

Still have subclasses for behavior



Solution - Strategy

Make a Strategy for each method of Component that varies in its subclasses.



Problem

But now instead of lots of component subclasses

We have lots of Strategy subclasses

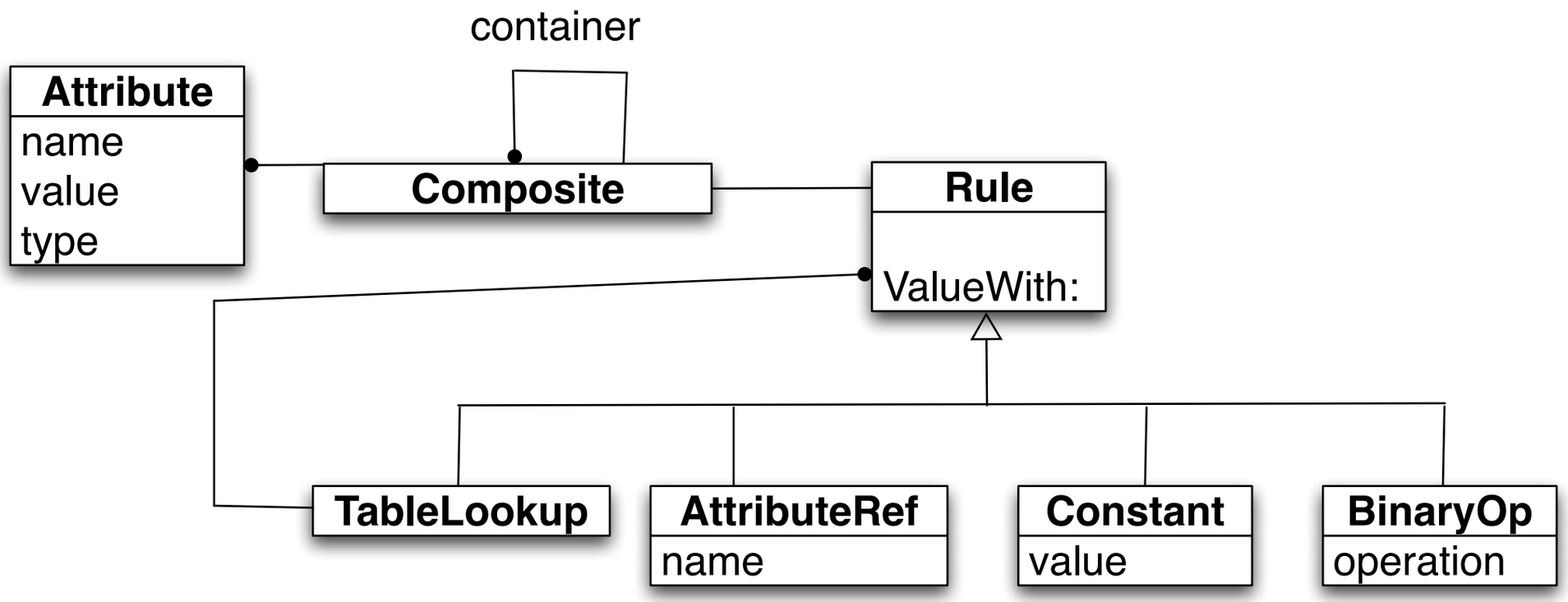
Solution - Interpreter

Create small language for the behaviors of strategies

Value strategies use:

- arithmetic expressions
- table look up
- if statements

Solution - Interpreter



Rules
- read/write attributes
- pre-formula
  - evaluated before component's children
- post-formula
  - evaluated after component's children

Problem

Component subclass replaced with attributes & rules

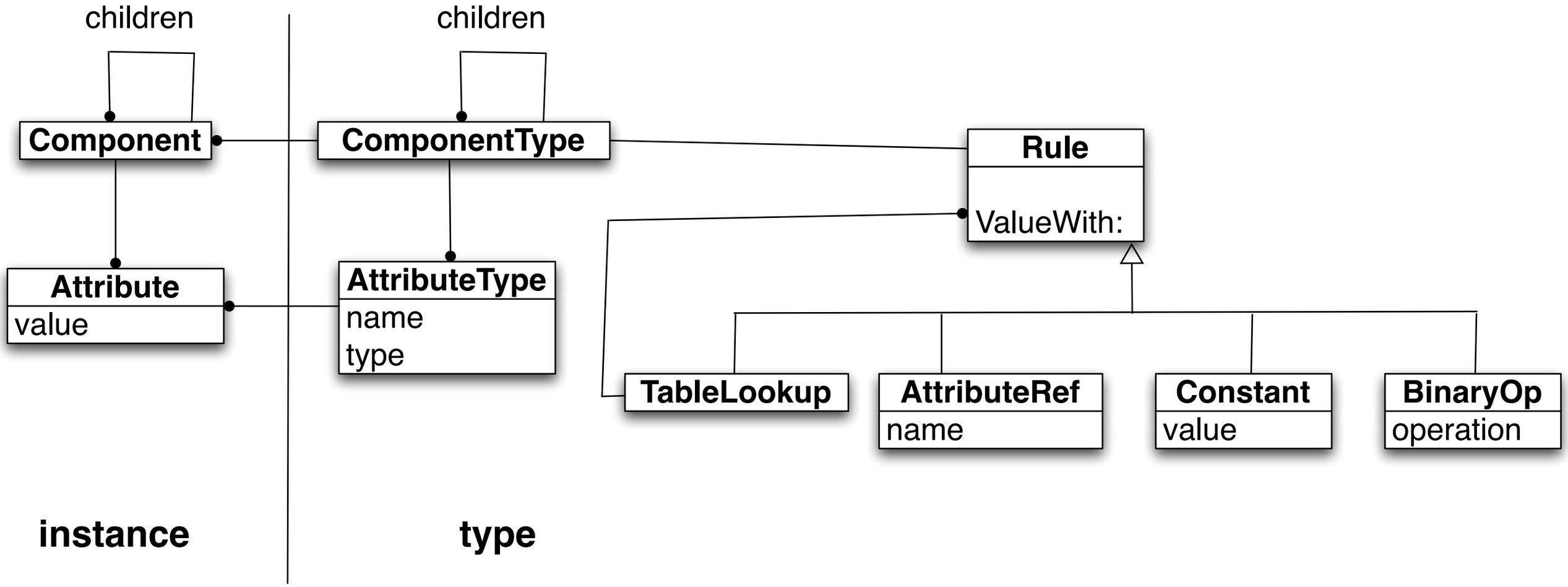
Each "component" instance has own copy of rules - duplication

Without classes to categorize components
  harder to understand code

How can you eliminate duplication in a component system and represent categories of similar components when all components have the same class?

Solution - Type Object

Use the Type Object pattern; i.e. make objects that represent the common features of a category of components, and let each component know its type and access those features by delegating to the type



Problem

Sometimes attributes need to have rules

Life insurance over $1,000,000 has special data and rules

Most attributes don't have rules so why add that option to all attributes

Solution - Decorator

AttributeDecorator - adds rule to attribute

Smart Variable

Issue

Often when a field changes some action is required

Most of the time accessor methods handle this fine

Examples when not

Debugger - watch points

Simulations

Real-time tracking of business

Actions tied to State Change

Dependent Notification
Persistence
Distribution
Caching
Constraint Satisfaction
Synchronization

Swift Property Observers

```
class PositiveTemperature {
    var degreesFarenheit: Double = 0 {
        willSet(newDegree) {
            print("Changing the temperature")
        }

        didSet {
            if degreesFarenheit < 0 {
                degreesFarenheit = oldValue
            }
        }
    }
}
```

```
var test = PositiveTemperature()
test.degreesFarenheit = 10        // Changing the temperature
test.degreesFarenheit             // 10
test.degreesFarenheit = -20       // Changing the temperature
test.degreesFarenheit             // 10
```

Schema

Schema

Descriptor
Map
Database Scheme
Layout

How do you avoid hard-wiring the layouts of structures into your code?
How do you describe the layout of a structure, object, or database row?

Therefore, make a schema or map describing your data structures available at runtime

Participants

Schema - collection of descriptors

Descriptor - describe layout of element
  May contain attributes
    display name, type, default value

Subject - objects being mapped by schema

Grapples - map between symbolic name to actual object

Attributes

Examples

Database Object-Relational mapping

Hibernate, Spring, Active Record in Ruby on Rails

GUI Builders

JavaBeans - Descriptor

GraphQL

Active Object Model

Active Object Model

Object model that provides “meta” information about itself
so that it can be changed at runtime

Why

Both systems and their users must adapt quickly to changing requirements

Dynamic Objects allow for rapid alterations to your program

Users want the ability to change what they do on-the-fly

Changing a program to meet new business requirements is slow and complicated

Problems

Active object-models can be
- difficult to develop
- hard to understand
- hard to maintain

So include editors and other tools
to assist with developing and manipulating the object model