2016 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

Design Patterns by Example for SystemVerilog Verification Environments Enabled by SystemVerilog 1800-2012

Eldon Nelson (eldon_nelson@ieee.org)

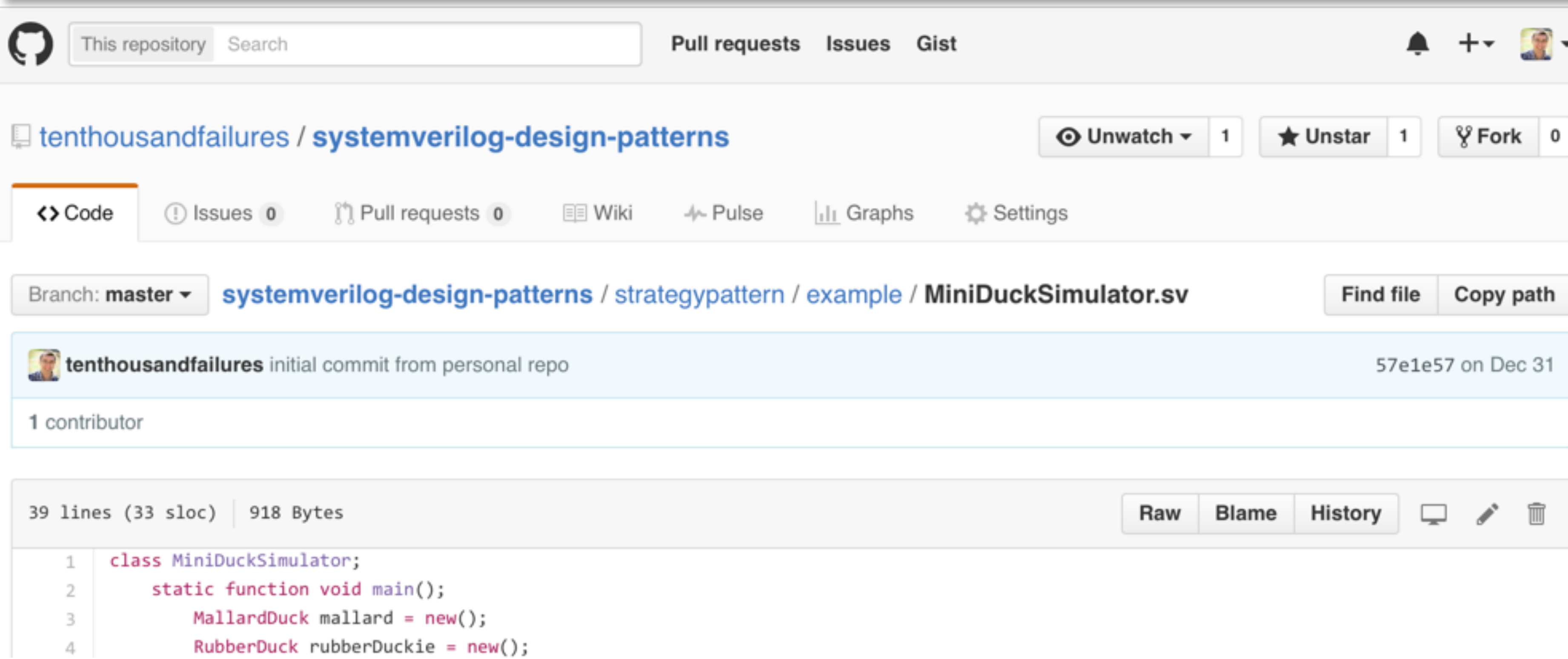
Intel Corporation

Outline

- Design Patterns
  - Approach
- Composition versus Inheritance
- Strategy Pattern
  - Class Interface and Implements
  - Application to Evolving Packet Class
- Decorator Pattern
  - Typed Constructor Calls
  - Application to Layered Constraints

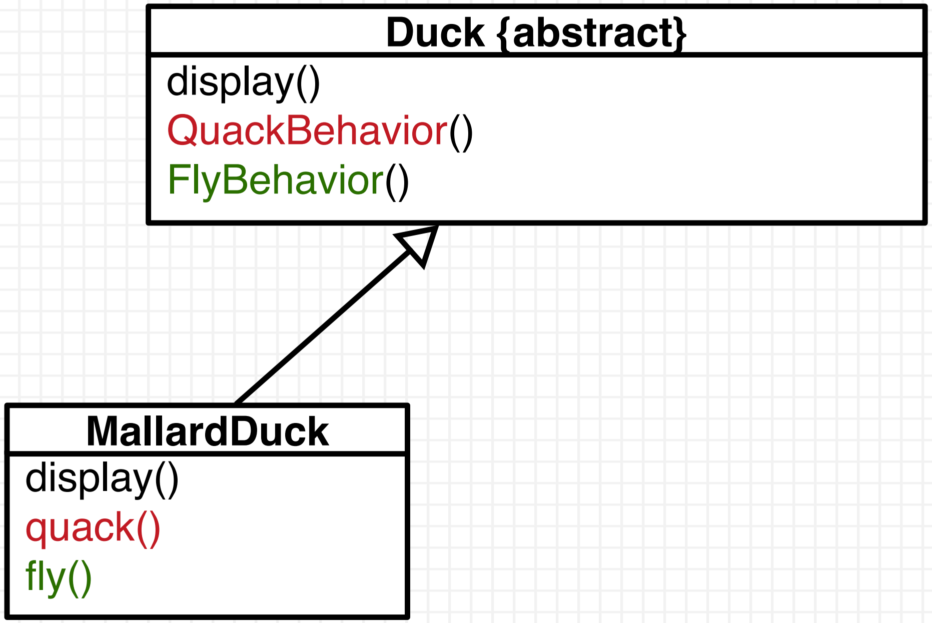
Full Code Examples

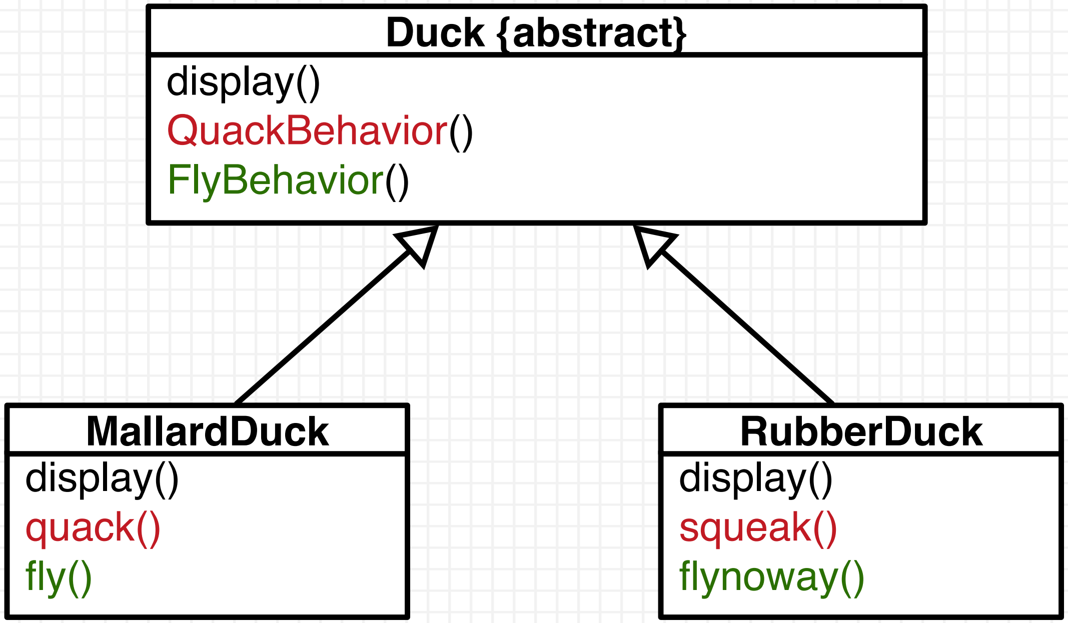
https://github.com/tenthousandfailures/systemverilog-design-patterns

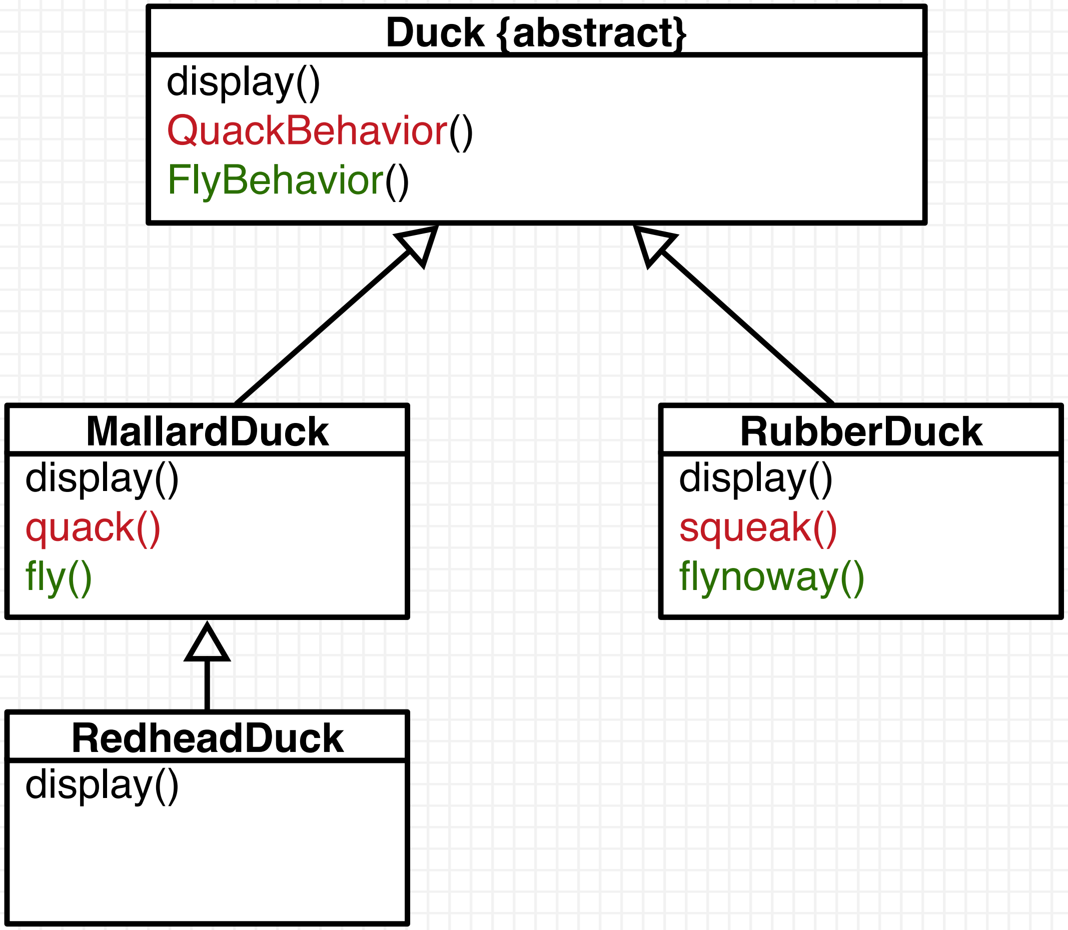


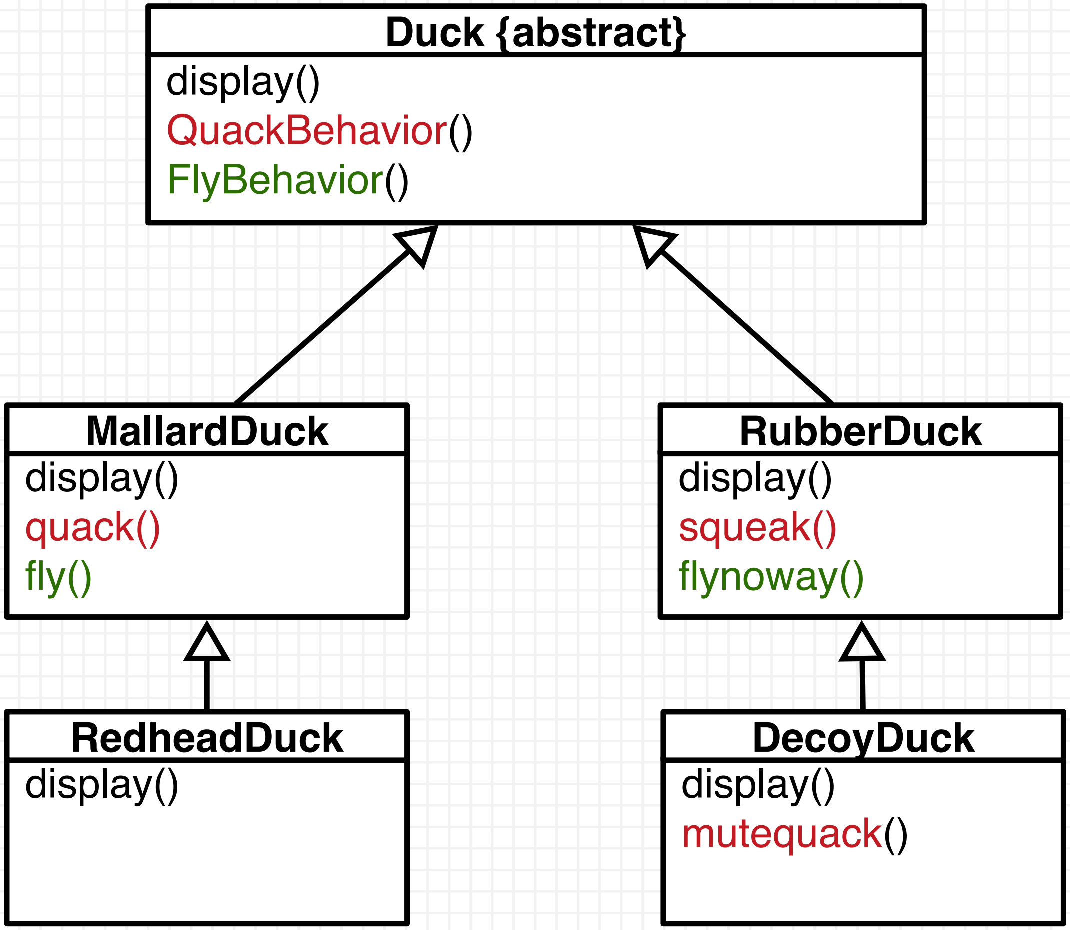
Duck {abstract}

display()
QuackBehavior()
FlyBehavior()

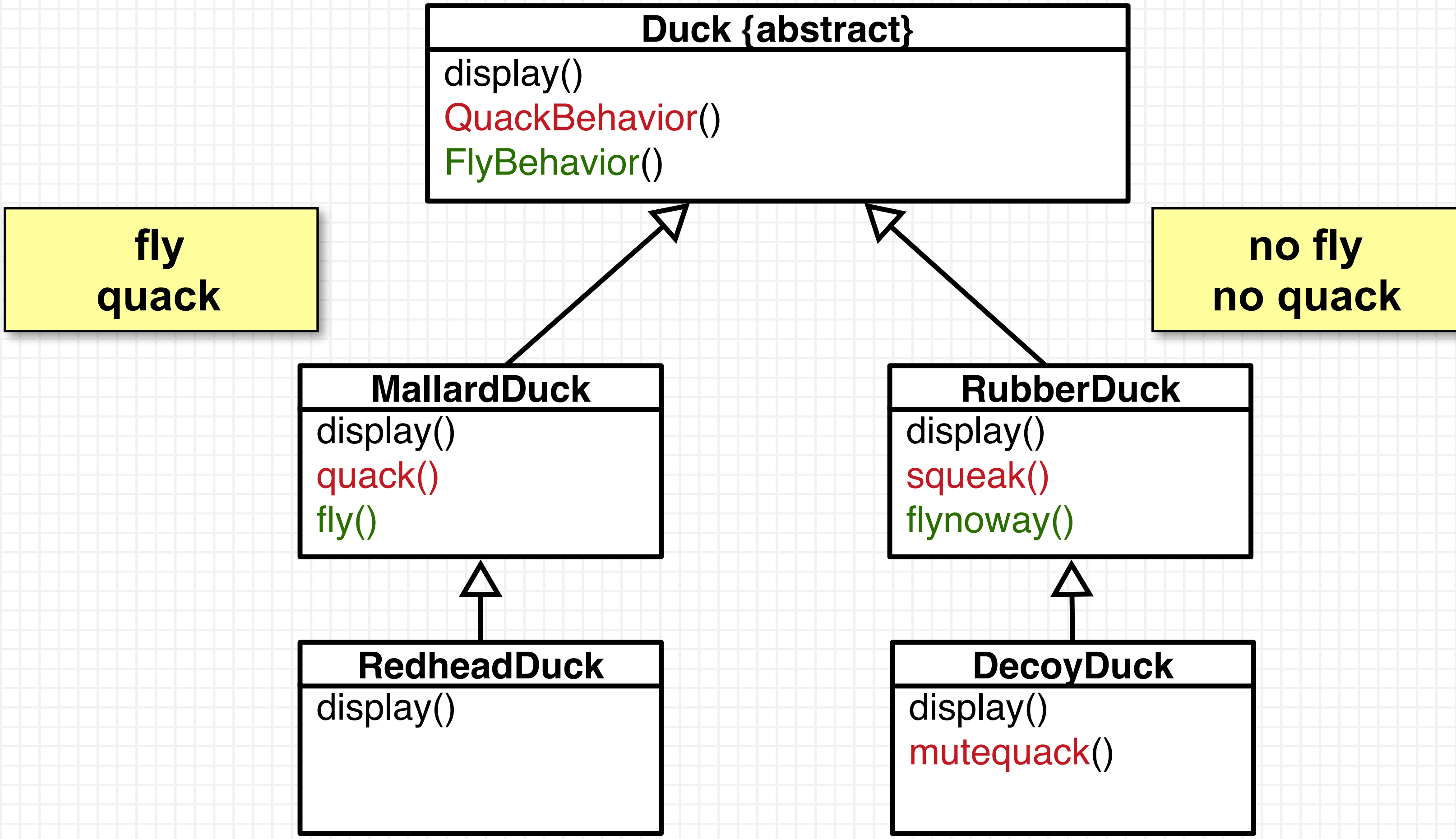

Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
MallardDuck
display()
quack()
fly()


Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()


Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()
RedheadDuck
display()

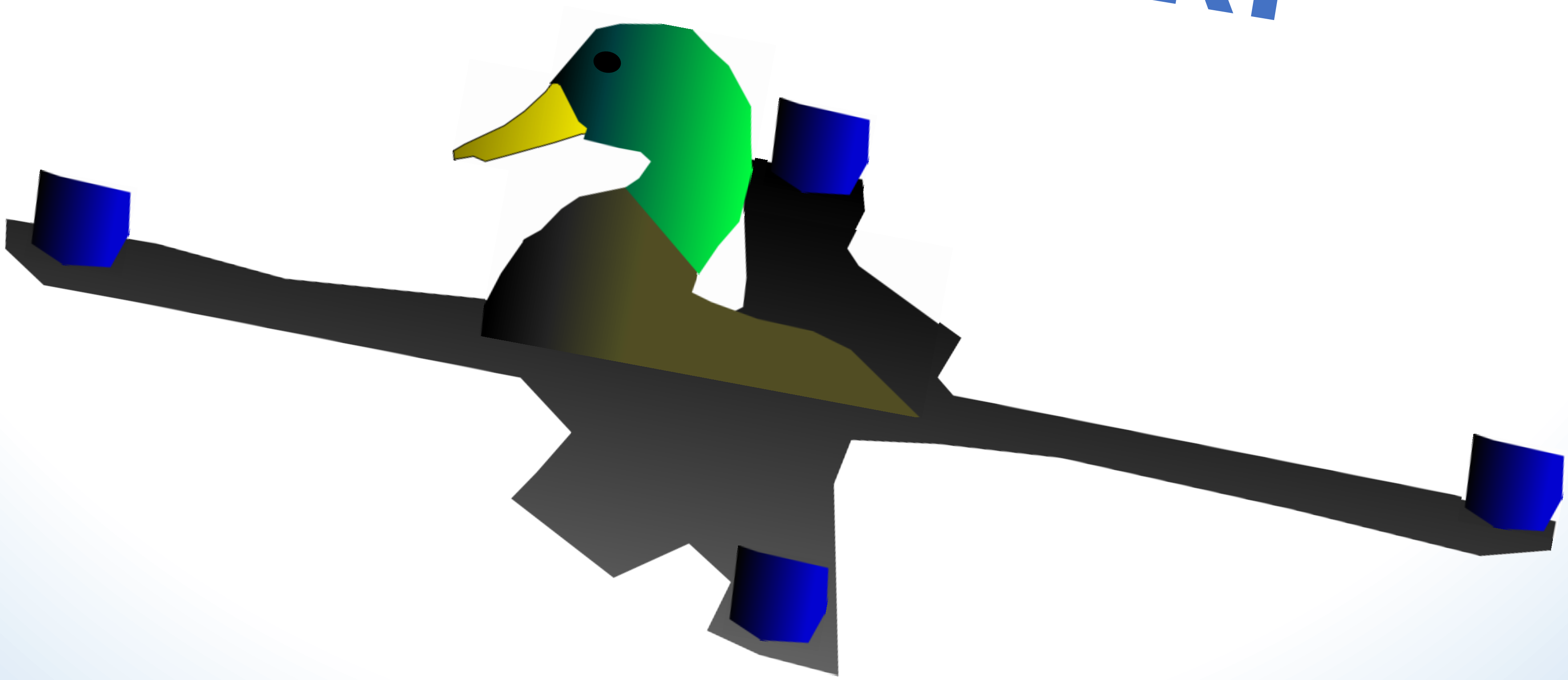


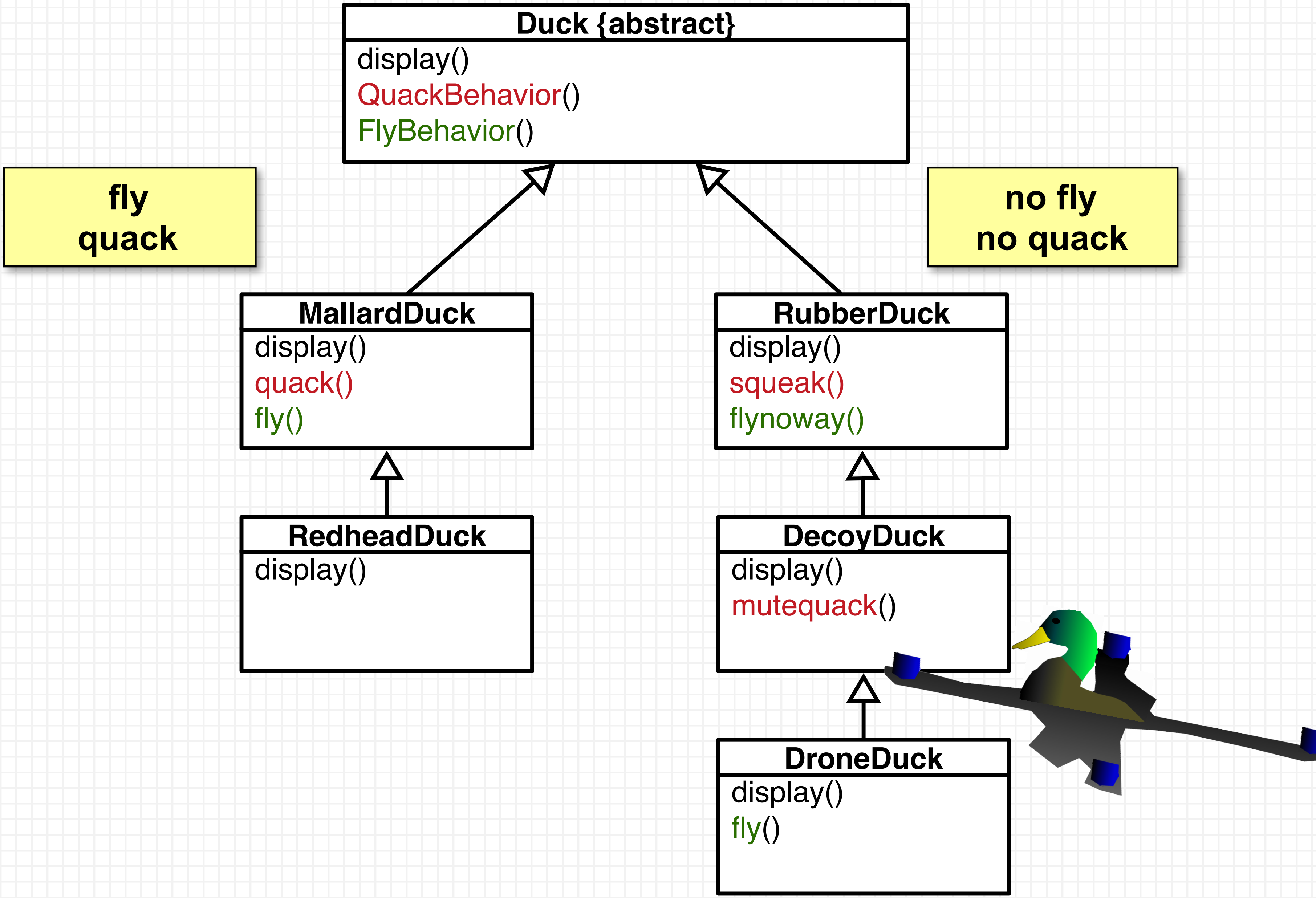
Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()
RedheadDuck
display()
DecoyDuck
display()
mutequack()

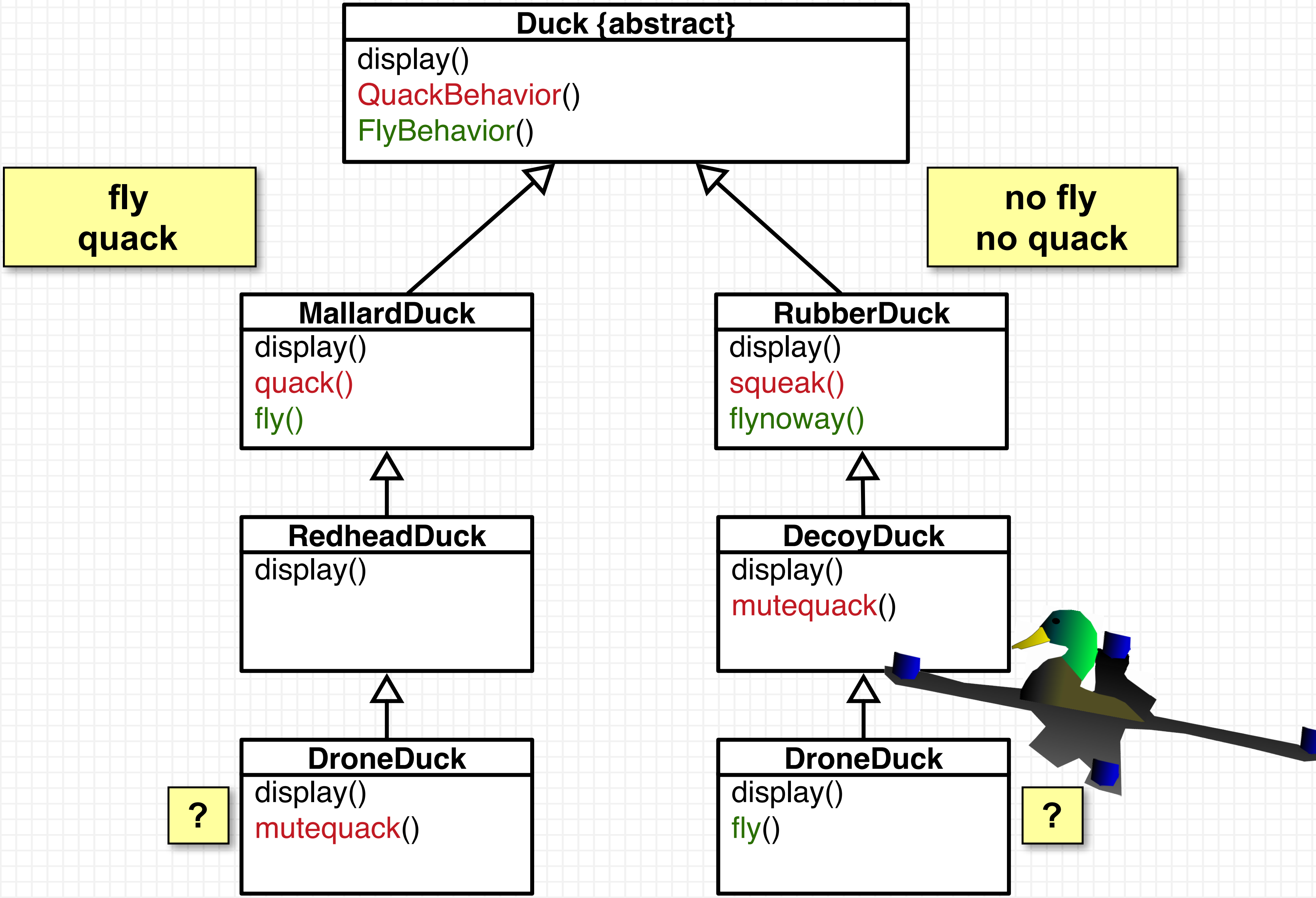


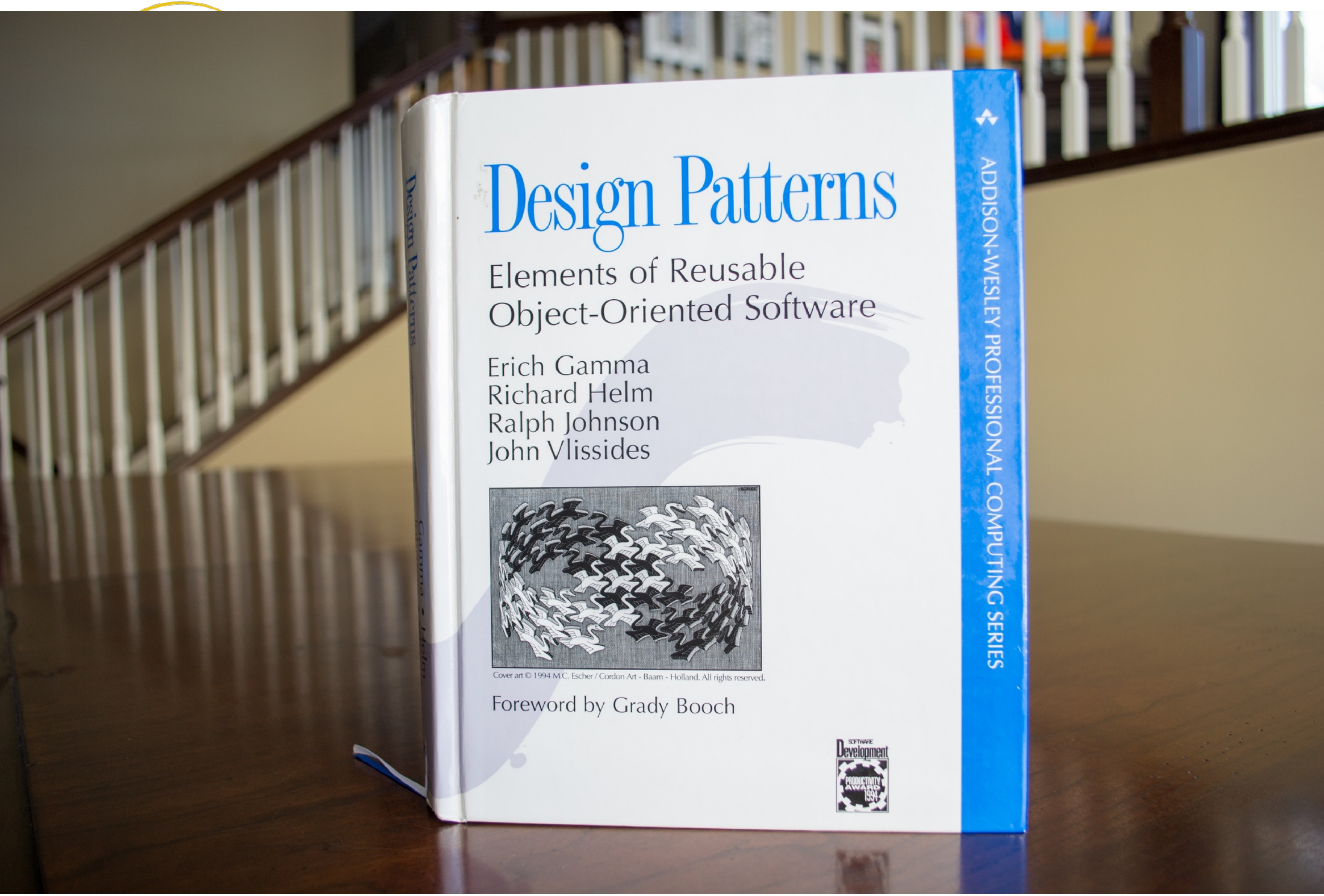
fly
quack
Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
no fly
no quack
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()
RedheadDuck
display()
DecoyDuck
display()
mutequack()

DroneDuck!



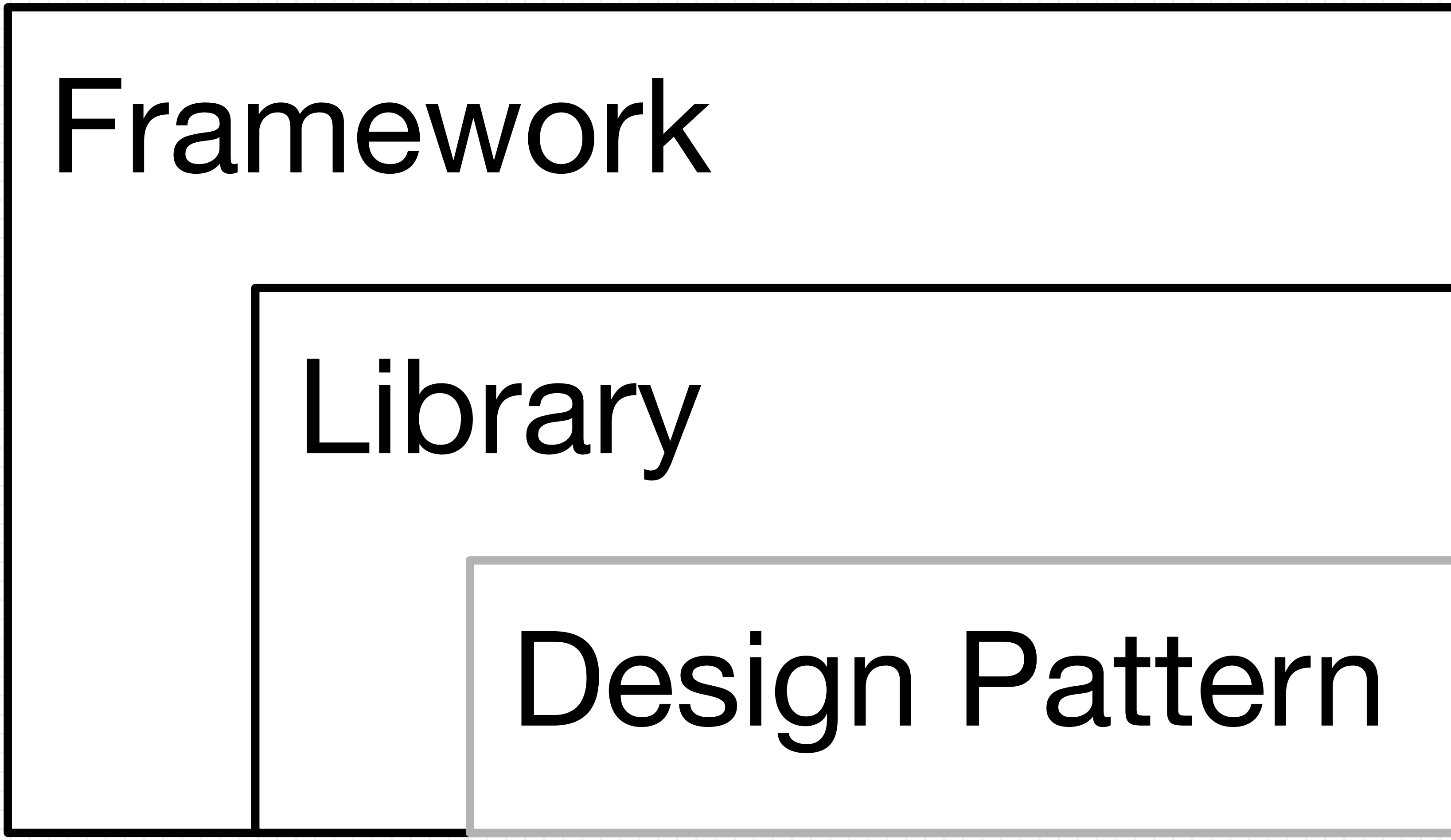

Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
fly
quack
no fly
no quack
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()
RedheadDuck
display()
DecoyDuck
display()
mutequack()
DroneDuck
display()
fly()

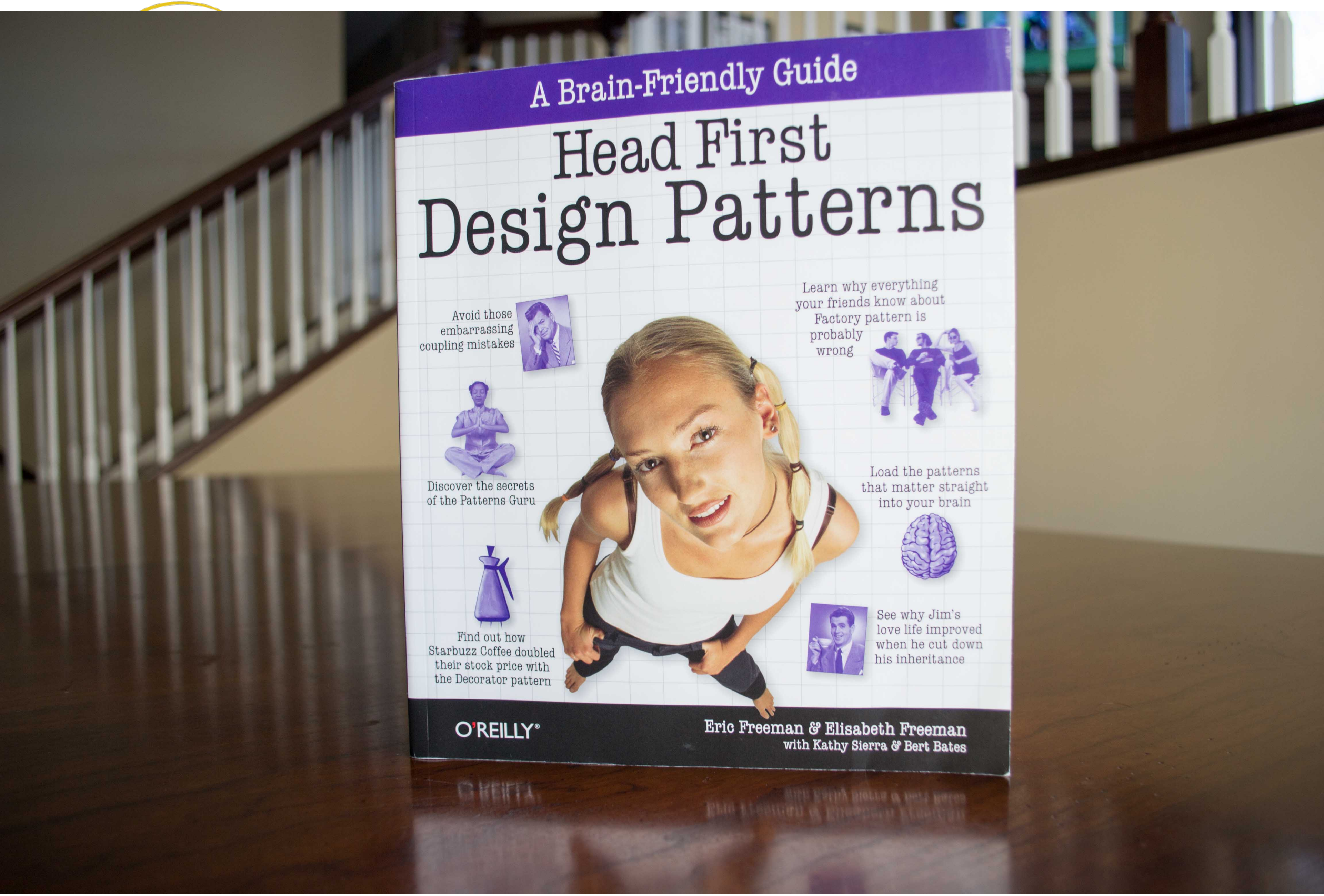

Duck {abstract}
display()
QuackBehavior()
FlyBehavior()
fly
quack
no fly
no quack
MallardDuck
display()
quack()
fly()
RubberDuck
display()
squeak()
flynoway()
RedheadDuck
display()
DecoyDuck
display()
mutequack()
DroneDuck
display()
mutequack()
?
DroneDuck
display()
fly()
?


Design Patterns
Elements of Reusable
Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
Design Patterns

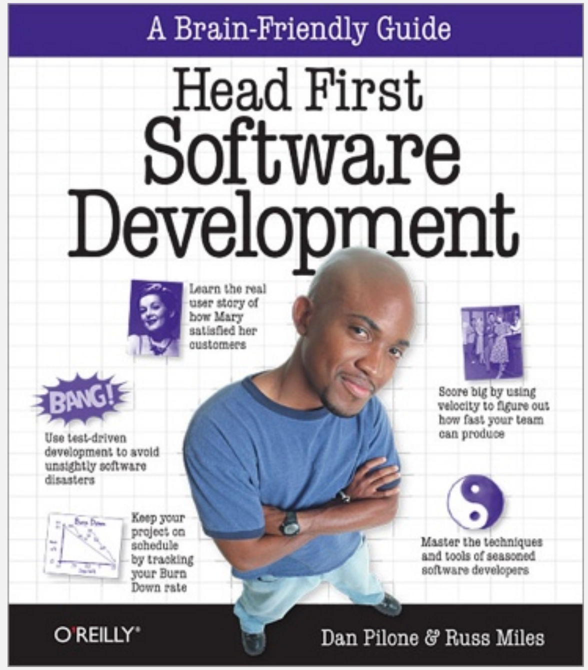
Design Patterns 1994

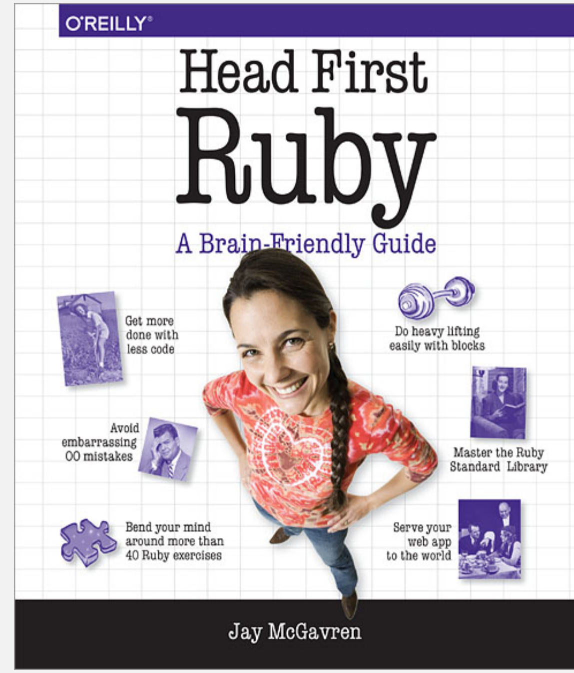
- Is the definitive first text
- Examples in the programming language Smalltalk or in C++ with features from 1994
- Code fragment examples center around a Word Processor GUI Application

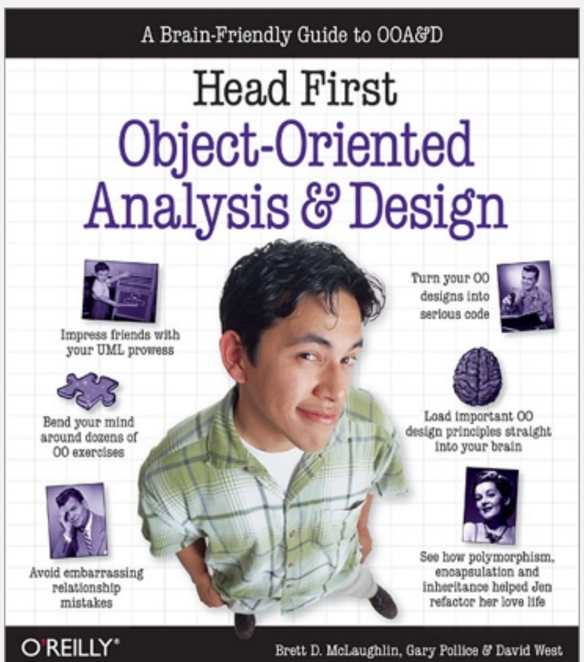

Framework
Library
Design Pattern

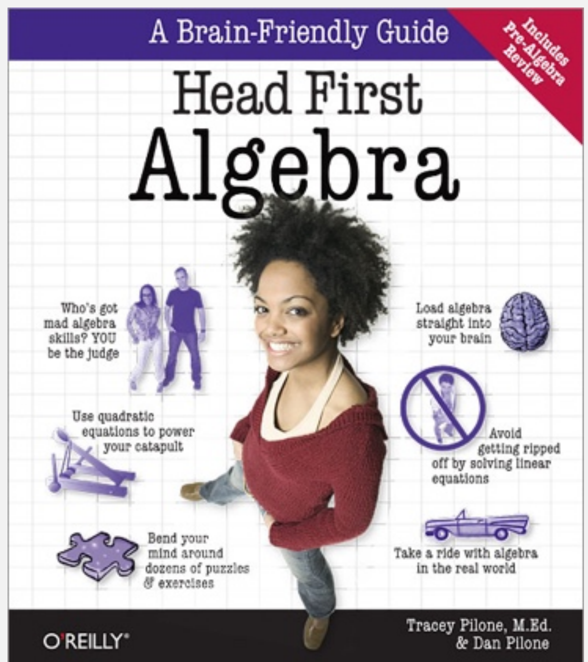

A Brain-Friendly Guide
Head First
Design Patterns
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
O'REILLY
Eric Freeman & Elisabeth Freeman
with Kathy Sierra & Bert Bates

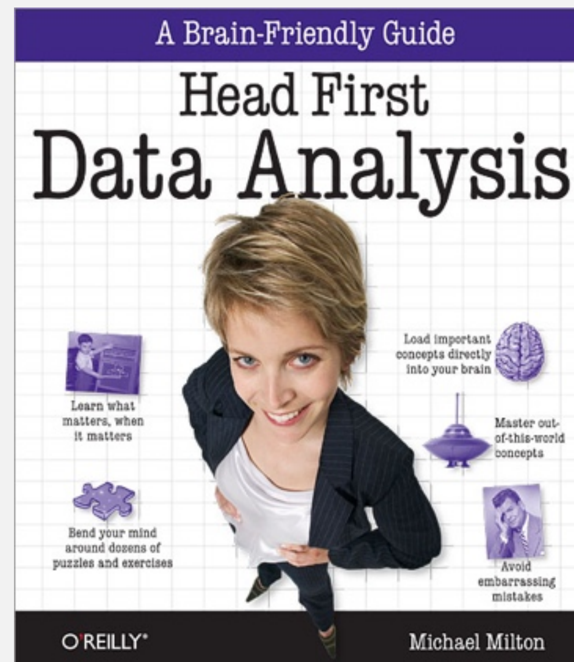

O'REILLY
10th Anniversary
Updated for Java 8
Head First
Design Patterns
A Brain-Friendly Guide
Avoid those embarrassing coupling mistakes
Learn why everything your friends know about Factory pattern is probably wrong
Discover the secrets of the Patterns Guru
Load the patterns that matter straight into your brain
Find out how Starbuzz Coffee doubled their stock price with the Decorator pattern
See why Jim's love life improved when he cut down his inheritance
Eric Freeman & Elisabeth Robson
with Kathy Sierra & Bert Bates


A Brain-Friendly Guide
Head First
Software
Development
Learn the real user story of how Mary satisfied her customers
BANG!
Use test-driven development to avoid unsightly software disasters
Score big by using velocity to figure out how fast your team can produce
Keep your project on schedule by tracking your Burn Down rate
Master the techniques and tools of seasoned software developers
O'REILLY
Dan Pilone & Russ Miles


O'REILLY
Head First
Ruby
A Brain-Friendly Guide
Get more done with less code
Do heavy lifting easily with blocks
Avoid embarrassing OO mistakes
Master the Ruby Standard Library
Bend your mind around more than 40 Ruby exercises
Serve your web app to the world
Jay McGavren


A Brain-Friendly Guide to OOA&D
Head First
Object-Oriented
Analysis & Design
Turn your OO designs into serious code
Impress friends with your UML prowess
Bend your mind around dozens of OO exercises
Load important OO design principles straight into your brain
Avoid embarrassing relationship mistakes
See how polymorphism, encapsulation and inheritance helped Jen refactor her love life
O'REILLY
Brett D. McLaughlin, Gary Pollice & David West


A Brain-Friendly Guide
Includes Pre-Algebra Review
Head First
Algebra
Who's got mad algebra skills? YOU be the judge
Load algebra straight into your brain
Use quadratic equations to power your catapult
Avoid getting ripped off by solving linear equations
Bend your mind around dozens of puzzles & exercises
Take a ride with algebra in the real world
O'REILLY
Tracey Pilone, M.Ed. & Dan Pilone


A Brain-Friendly Guide
Head First
Data Analysis
Load important concepts directly into your brain
Learn what matters, when it matters
Master out-of-this-world concepts
Bend your mind around dozens of puzzles and exercises
Avoid embarrassing mistakes
O'REILLY
Michael Milton

Head First Design Patterns

- Examples written solely in Java
- Complete examples given for every exercise
- Examples are on a variety of whimsical fictional applications

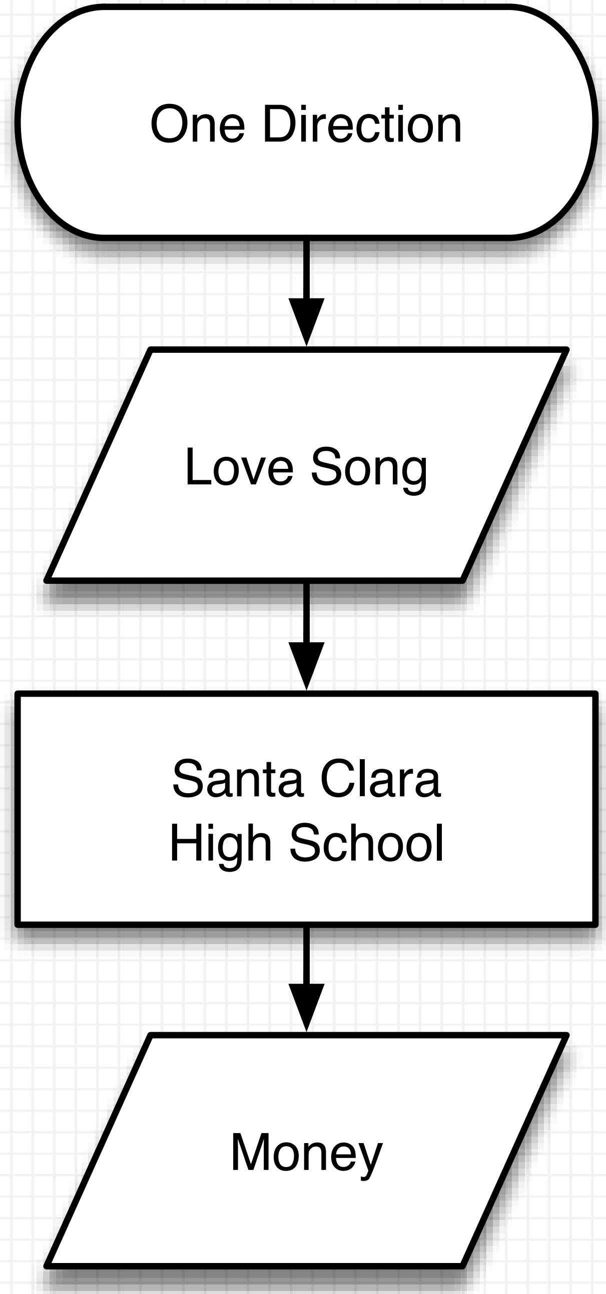
Outline

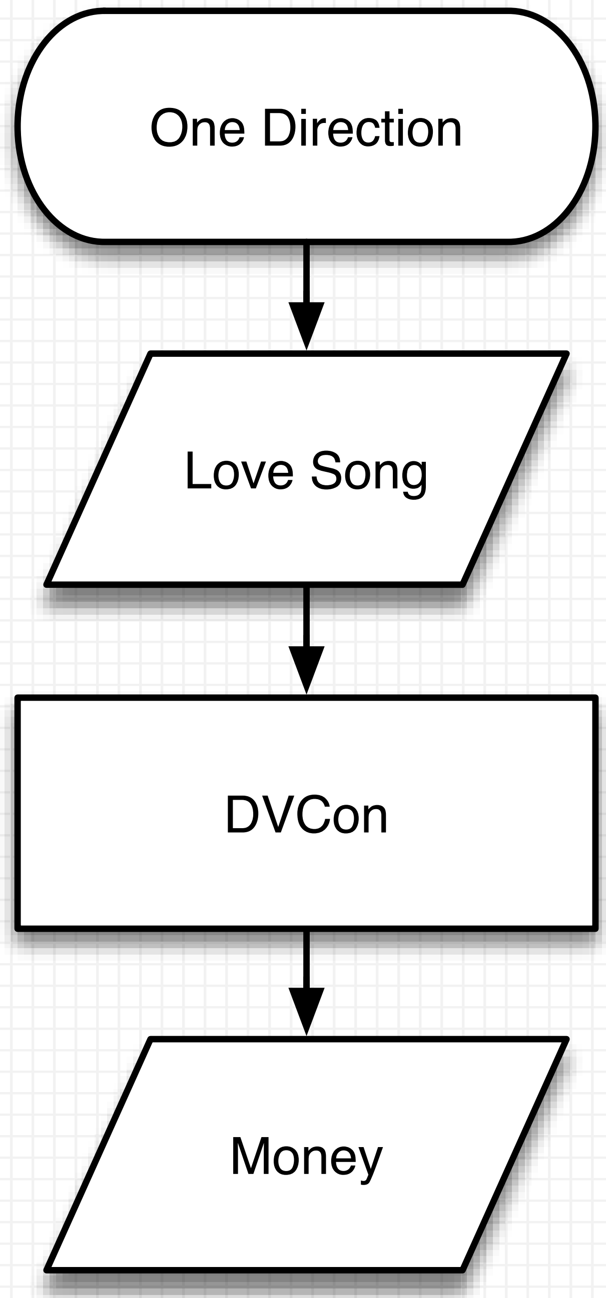
- Design Patterns
  - Approach
- **Composition versus Inheritance**
- Strategy Pattern
  - Class Interface and Implements
  - Application to Evolving Packet Class
- Decorator Pattern
  - Typed Constructor Calls
  - Application to Layered Constraints

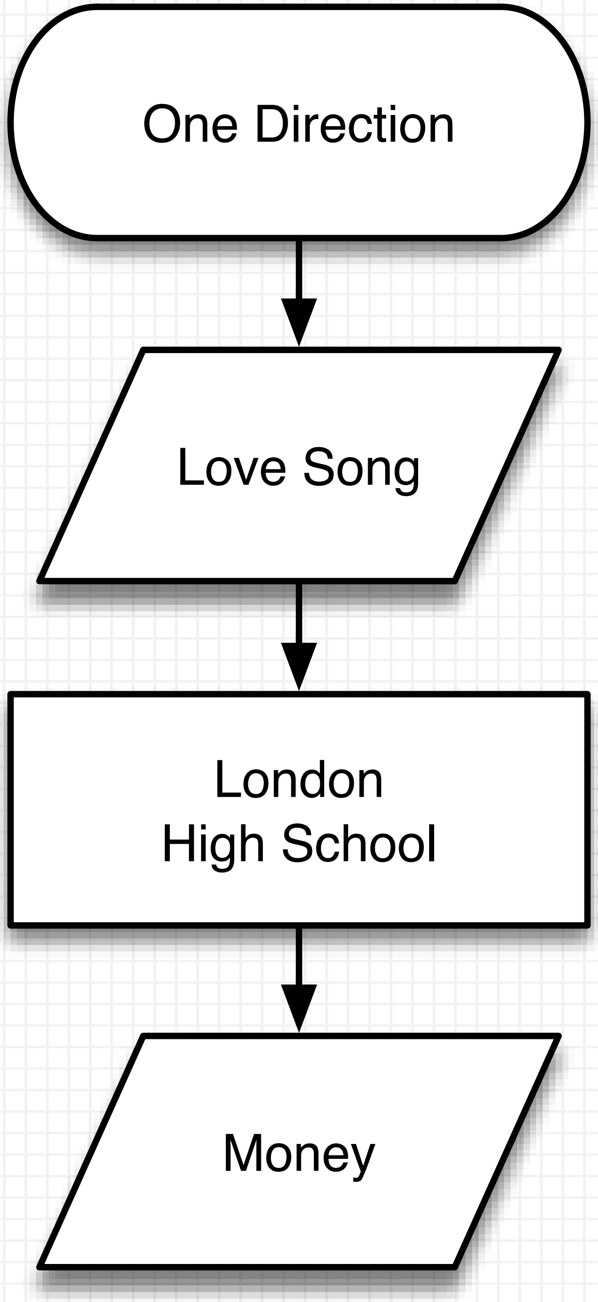
Composition

“Object composition is an alternative to class inheritance. Here, new functionality is obtained by assembling or composing objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. This style of reuse is called black-box reuse, because no internal details of objects are visible. Objects appear only as ‘black boxes’”

- “Design Patterns” p.19


One Direction
Love Song
Santa Clara
High School
Money

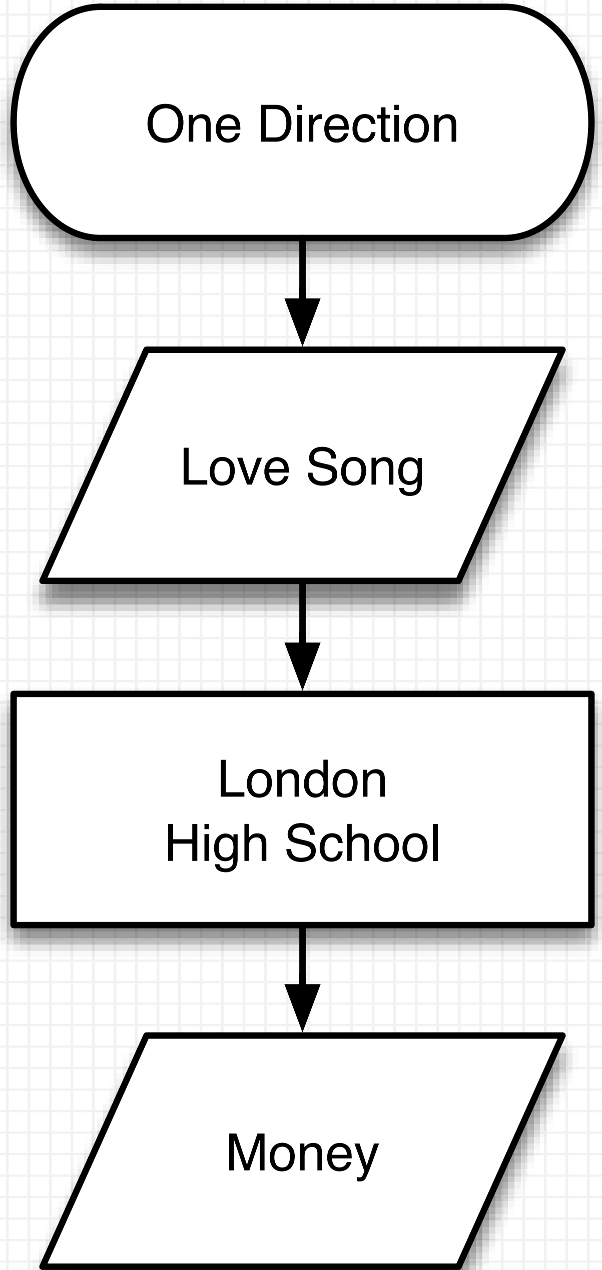

One Direction
Love Song
DVCon
Money

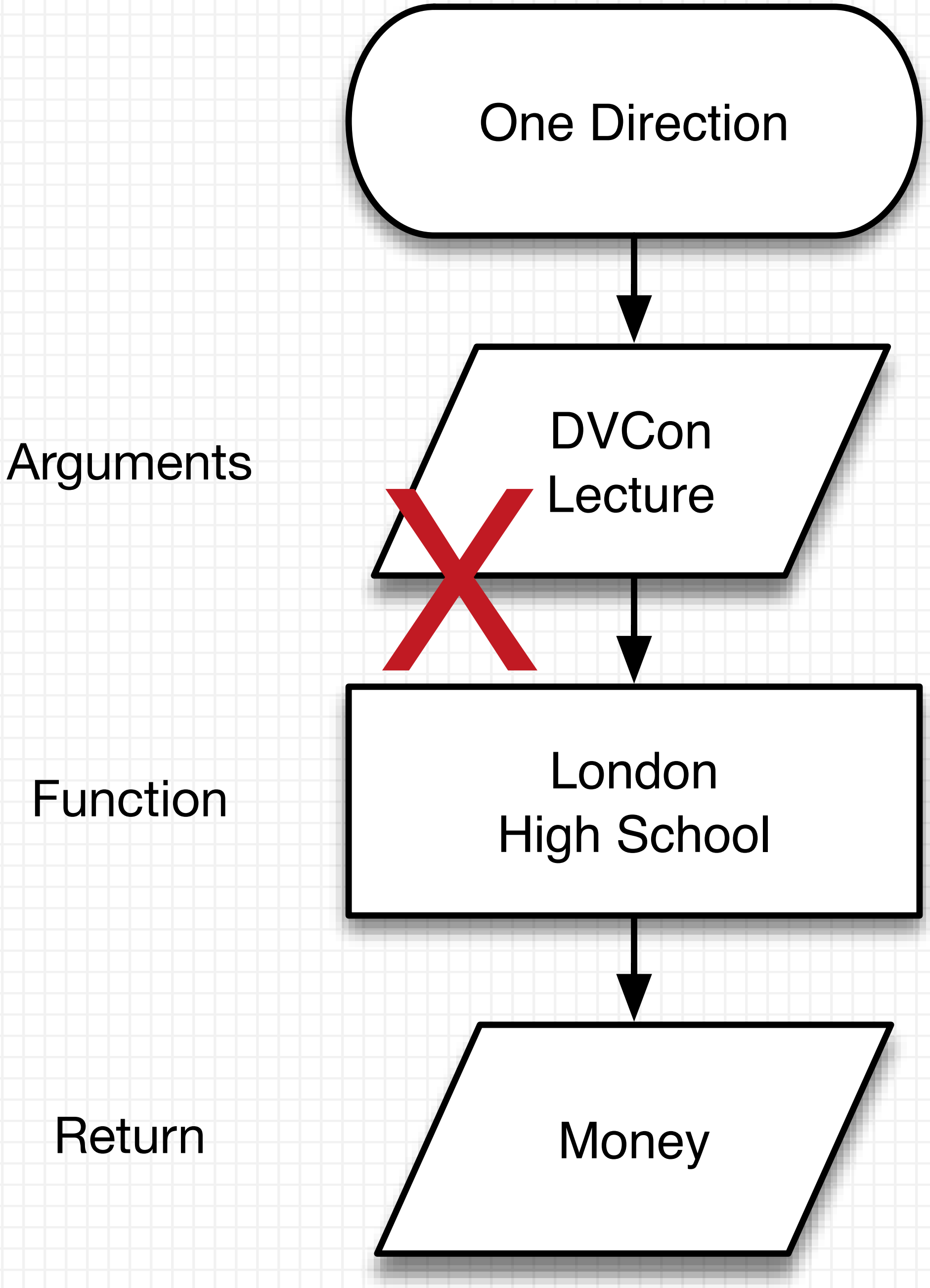

One Direction
Love Song
London
High School
Money

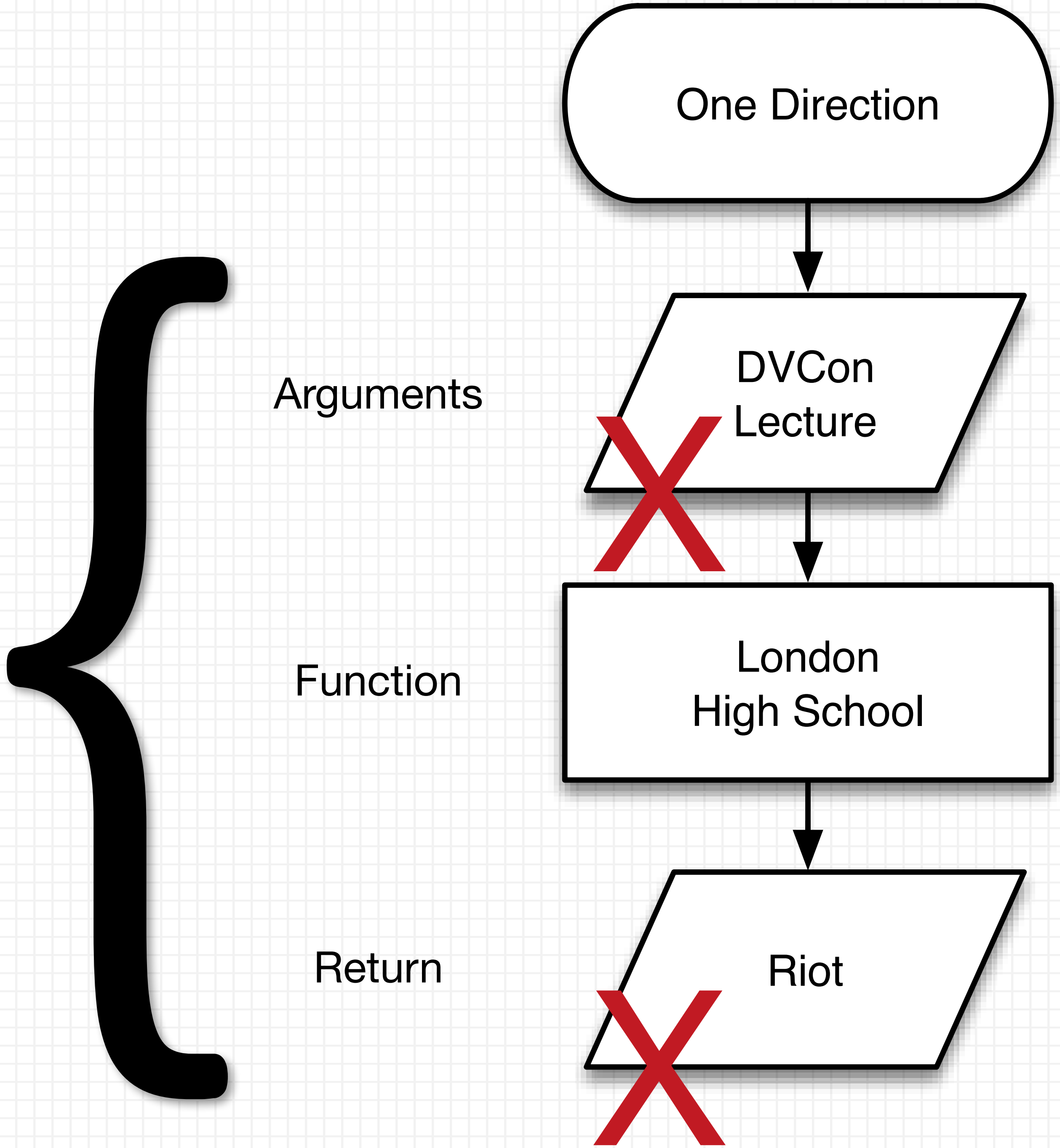
Arguments

Function

Return

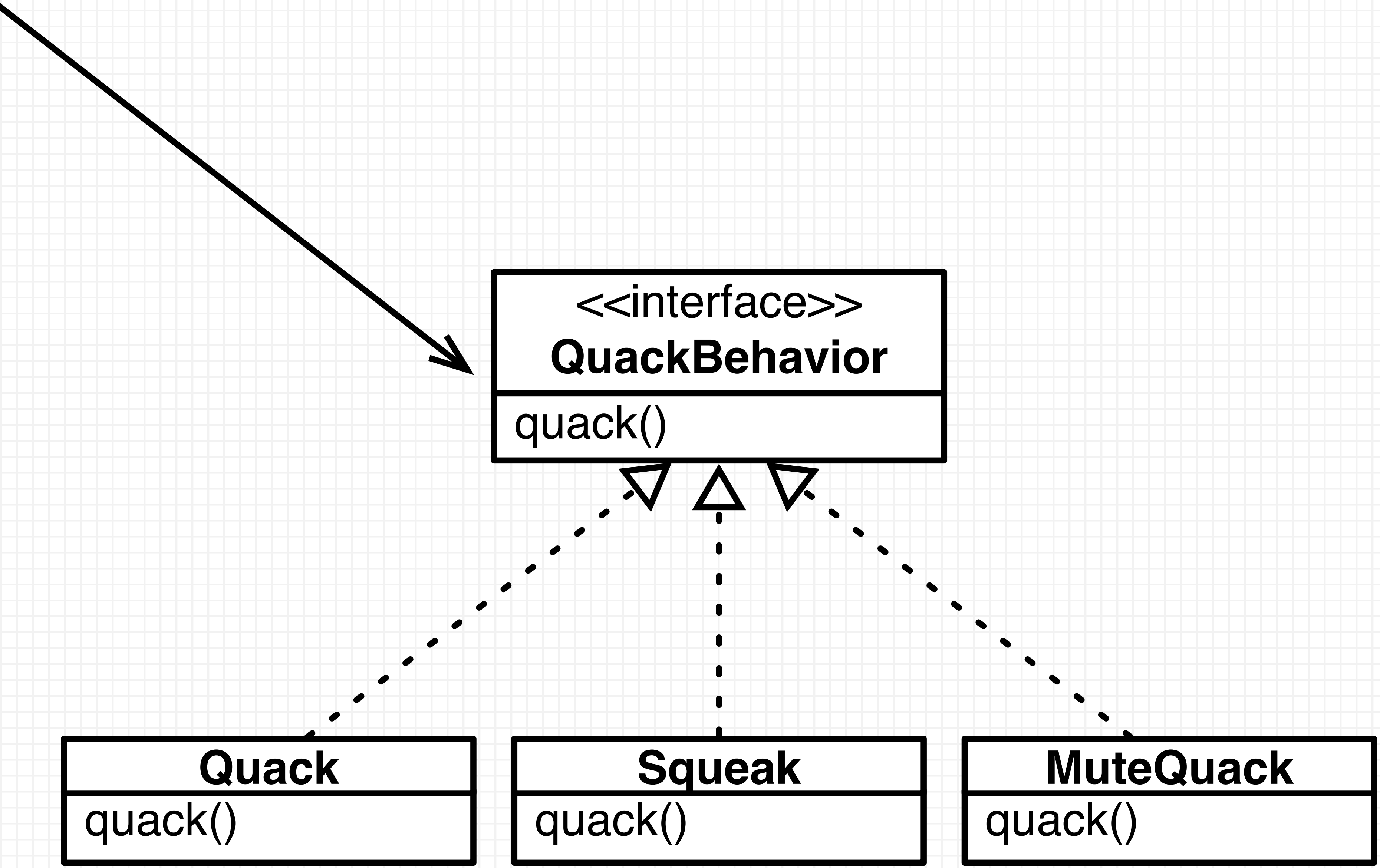



One Direction
Arguments
DVCon
Lecture
Function
London
High School
Return
Money


One Direction
Arguments
DVCon
Lecture
Function
London
High School
Return
Riot

Interface Class (Contract)

- “interface class” cannot define implementation
  - In contrast with an “virtual class” which can
  - Defines the arguments and return values
- “interface class” must have
  only “pure virtual function”
- No: constraints or covergroups


<<interface>>
QuackBehavior
quack()
Quack
quack()
Squeak
quack()
MuteQuack
quack()

Interface Class (Contract)

JAVA

```
public interface QuackBehavior {
    public void quack();
}
```

SYSTEMVERILOG

```
interface class QuackBehavior;
    pure virtual function void quack();
endclass
```

New SV
1800-2012

Implements

JAVA

```
public class Squeak implements QuackBehavior {
  public void quack() {
    System.out.println("Squeak");
    }
}
```

New SV
1800-2012

SYSTEMVERILOG

```
class Squeak implements QuackBehavior;
  virtual function void quack();
    $display("Squeak");
  endfunction
endclass
```

Phrase “Duck Typing”

- “In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.”

  – *Alexis Martelli, 7/26/2000 in comp.lang.python*

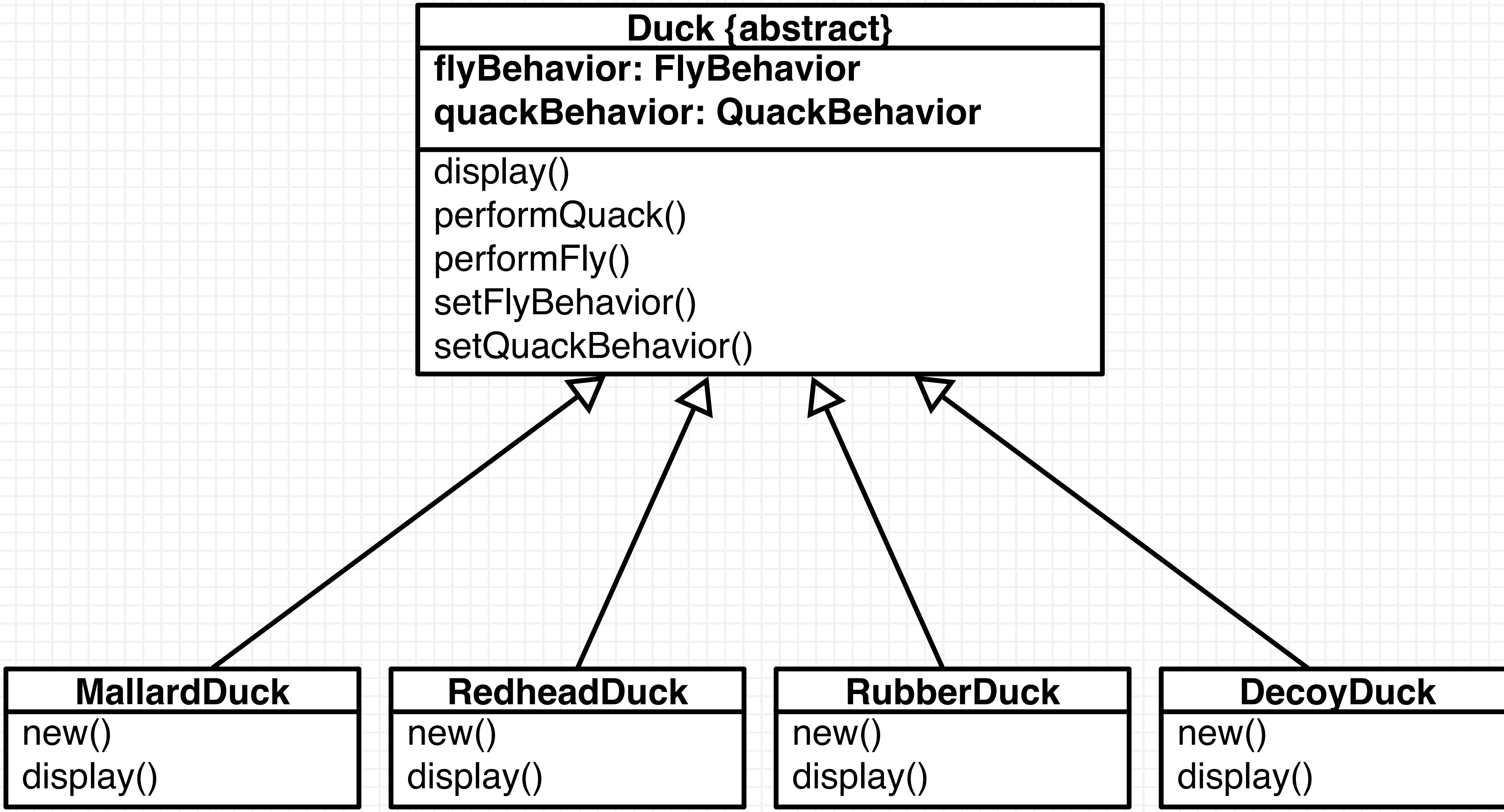
Outline

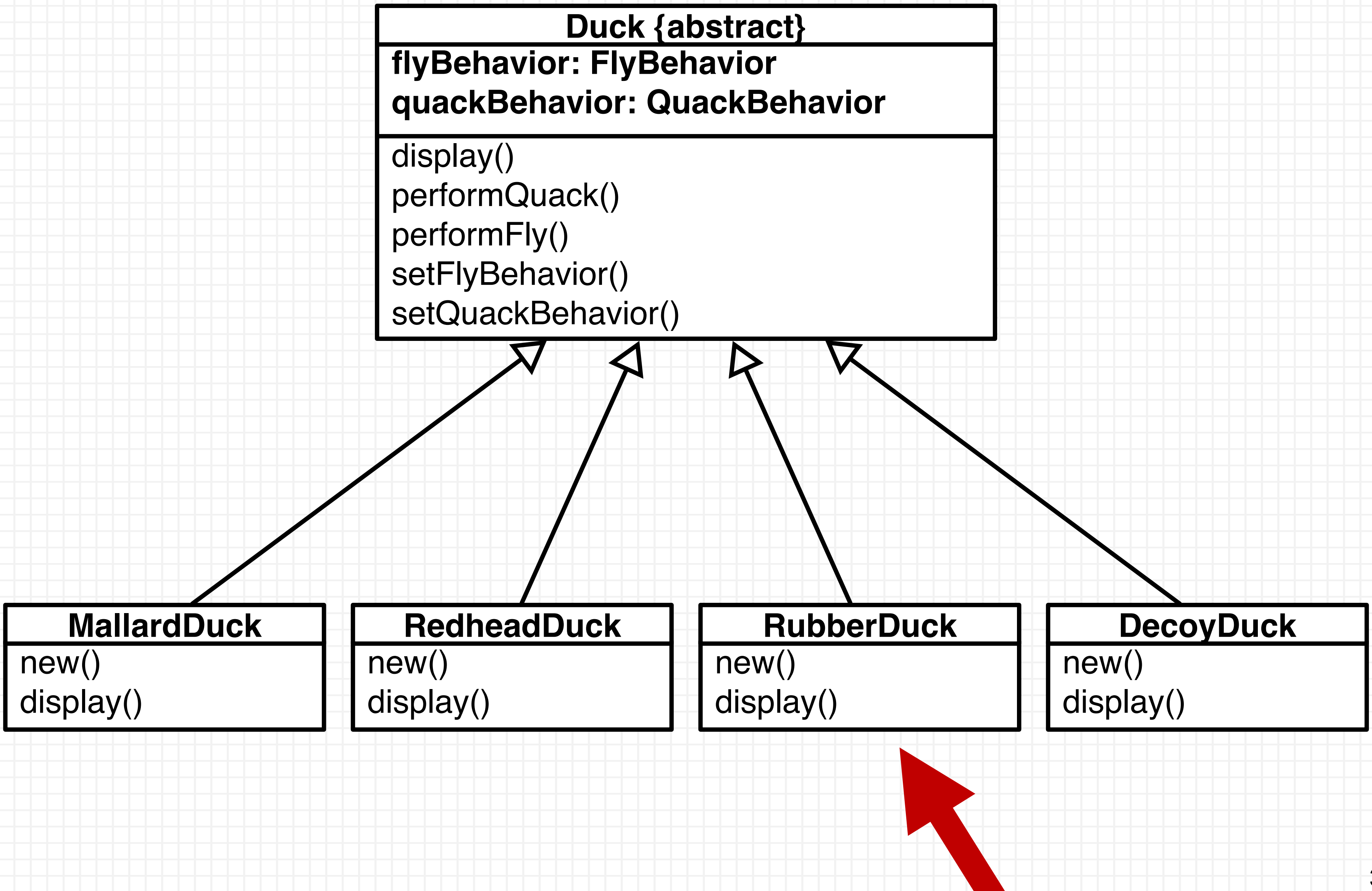
- Design Patterns
  - Approach
- Composition versus Inheritance
- **Strategy Pattern**
  - Class Interface and Implements
  - Application to Evolving Packet Class
- Decorator Pattern
  - Typed Constructor Calls
  - Application to Layered Constraints

Strategy Pattern

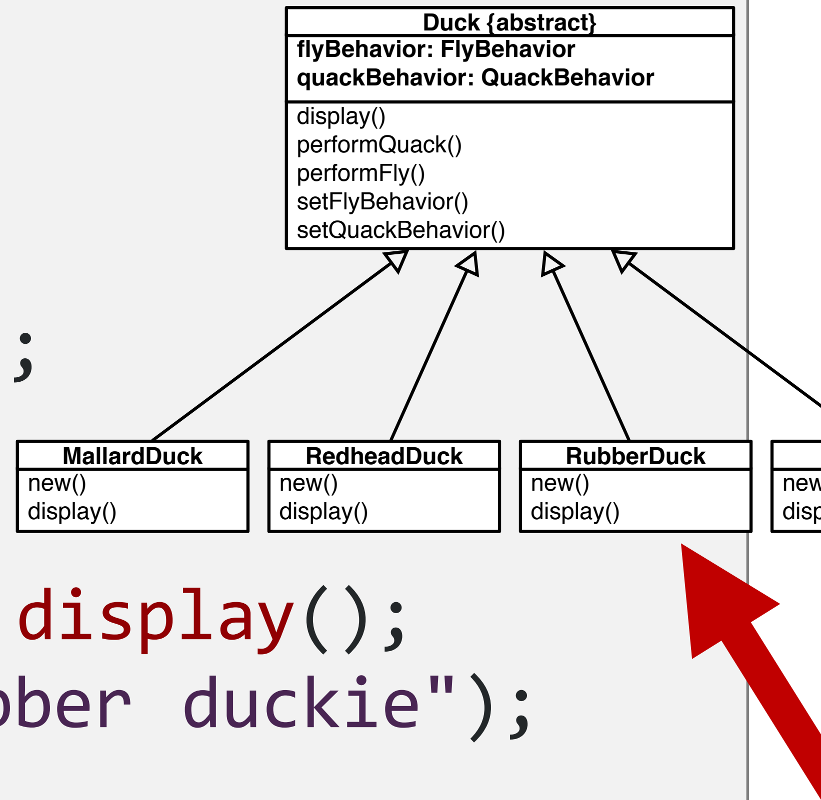
"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."

- Definition of the Strategy Pattern from "Design Patterns" [1, p. 135]

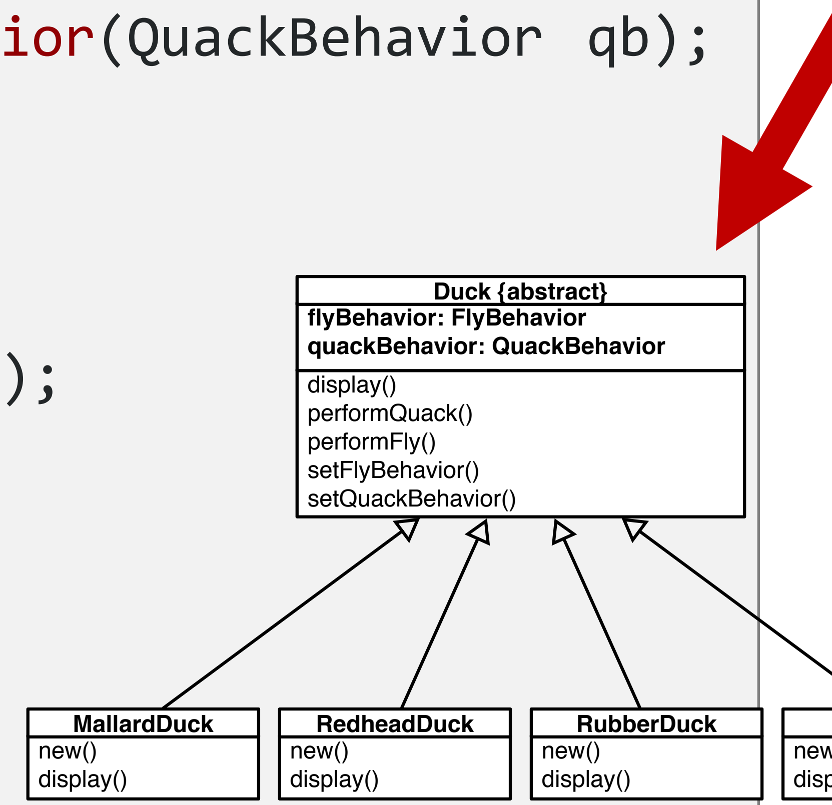

Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
MallardDuck
new()
display()
RedheadDuck
new()
display()
RubberDuck
new()
display()
DecoyDuck
new()
display()

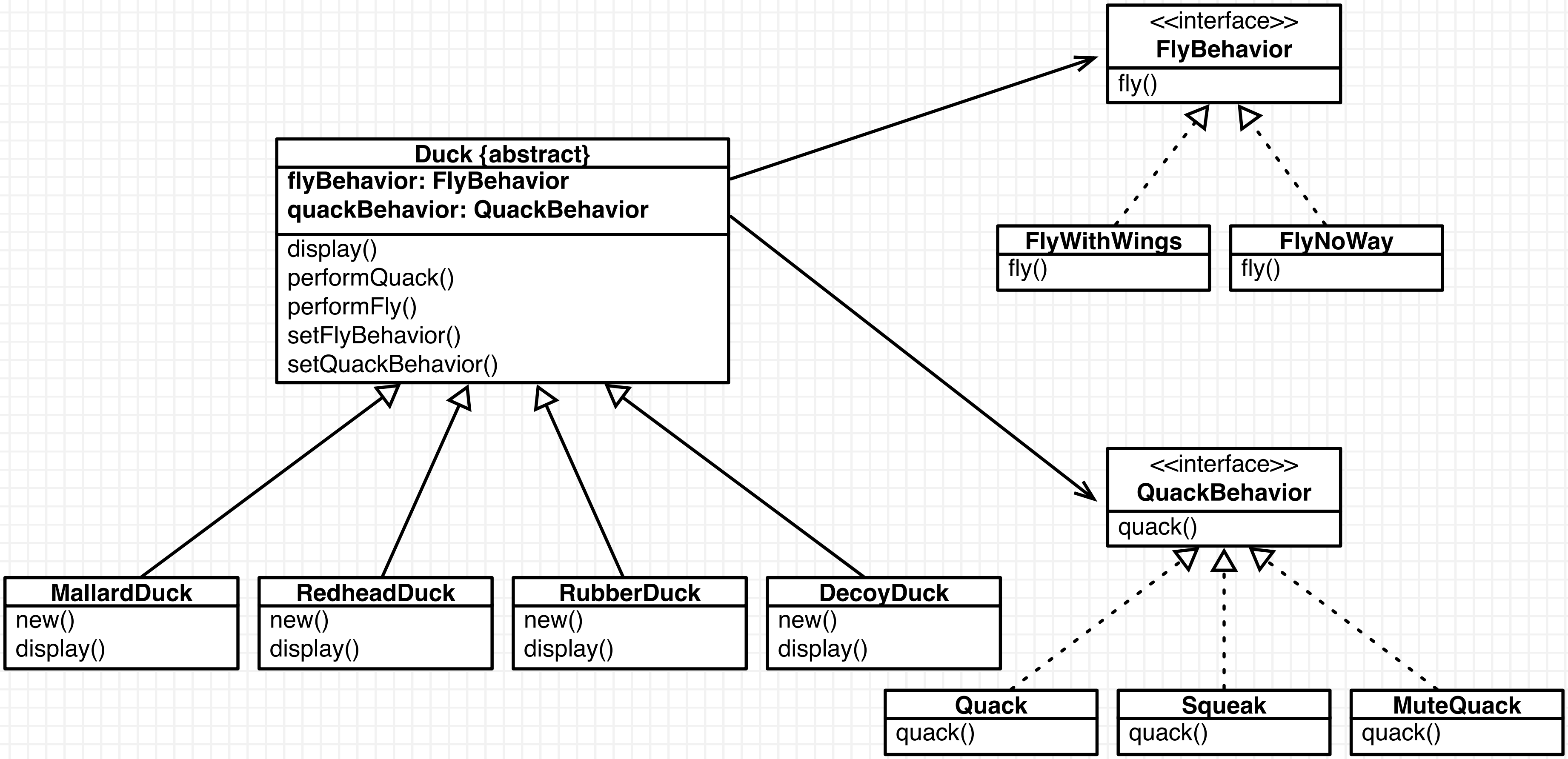

Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
MallardDuck
new()
display()
RedheadDuck
new()
display()
RubberDuck
new()
display()
DecoyDuck
new()
display()

```
class RubberDuck extends Duck;

   function new();
      FlyNoWay f = new();
      Squeak q = new();
      setFlyBehavior(f);
      setQuackBehavior(q);
   endfunction

   virtual function void display();
      $display("I'm a rubber duckie");
   endfunction

endclass
```



```
virtual class Duck;
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    function void setQuackBehavior(QuackBehavior qb);
        quackBehavior = qb;
    endfunction

    function void performQuack();
        quackBehavior.quack();
    endfunction
```





<<interface>>
FlyBehavior
fly()
Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
FlyWithWings
fly()
FlyNoWay
fly()
<<interface>>
QuackBehavior
quack()
MallardDuck
new()
display()
RedheadDuck
new()
display()
RubberDuck
new()
display()
DecoyDuck
new()
display()
Quack
quack()
Squeak
quack()
MuteQuack
quack()

Favor Composition

"Favor object composition over class inheritance."

- Second Principle of Object-Oriented Programming from "Design Patterns" [1, p. 20]

Strategy Pattern Applied to Packet Problem

- Encapsulation of a changing packet format
- Fields change width and location
- Changeable checking algorithm – CRC or Parity

Policy Classes AKA Strategy Pattern

Summary

Policy Classes

Each of UVM's policy classes perform a specific task for uvm_object-based objects: printing, comparing, recording, packing, and unpacking.

Outline

- Design Patterns
  - Approach
- Composition versus Inheritance
- Strategy Pattern
  - Class Interface and Implements
  - Application to Evolving Packet Class
- **Decorator Pattern**
  - Typed Constructor Calls
  - Application to Layered Constraints

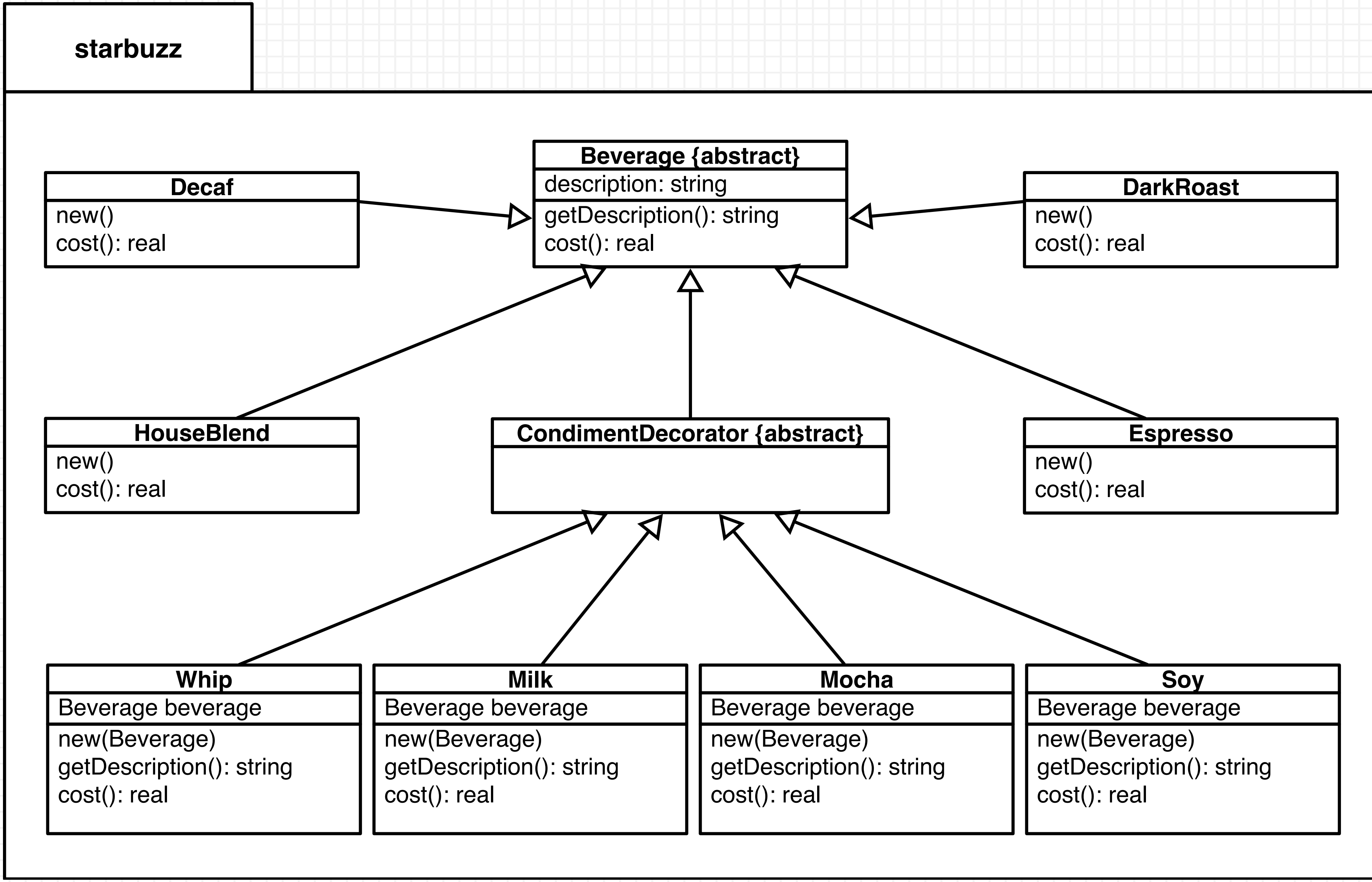
Decorator Pattern

"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extended functionality."

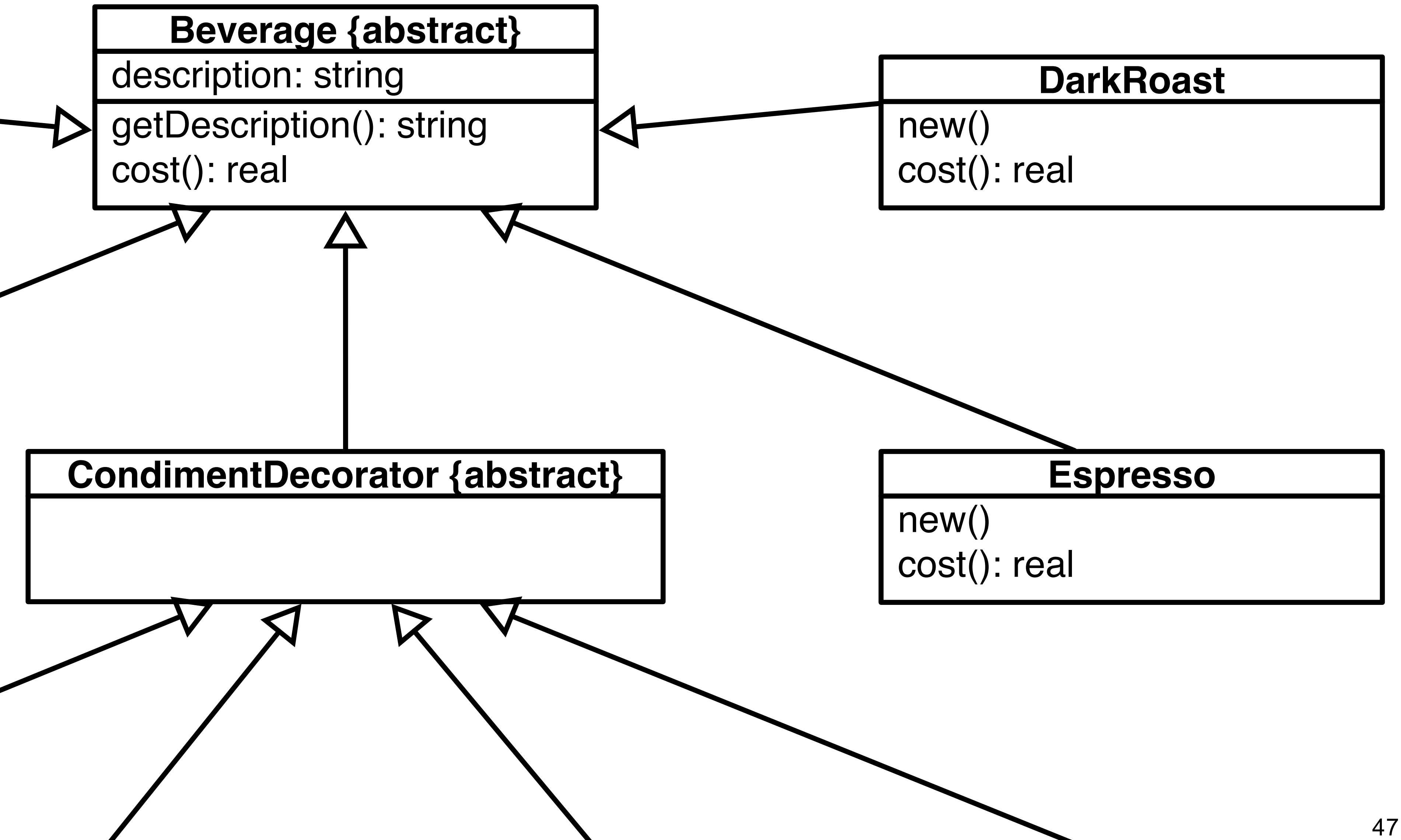
- *Definition of the Decorator Pattern from "Design Patterns" [1, p. 175]*

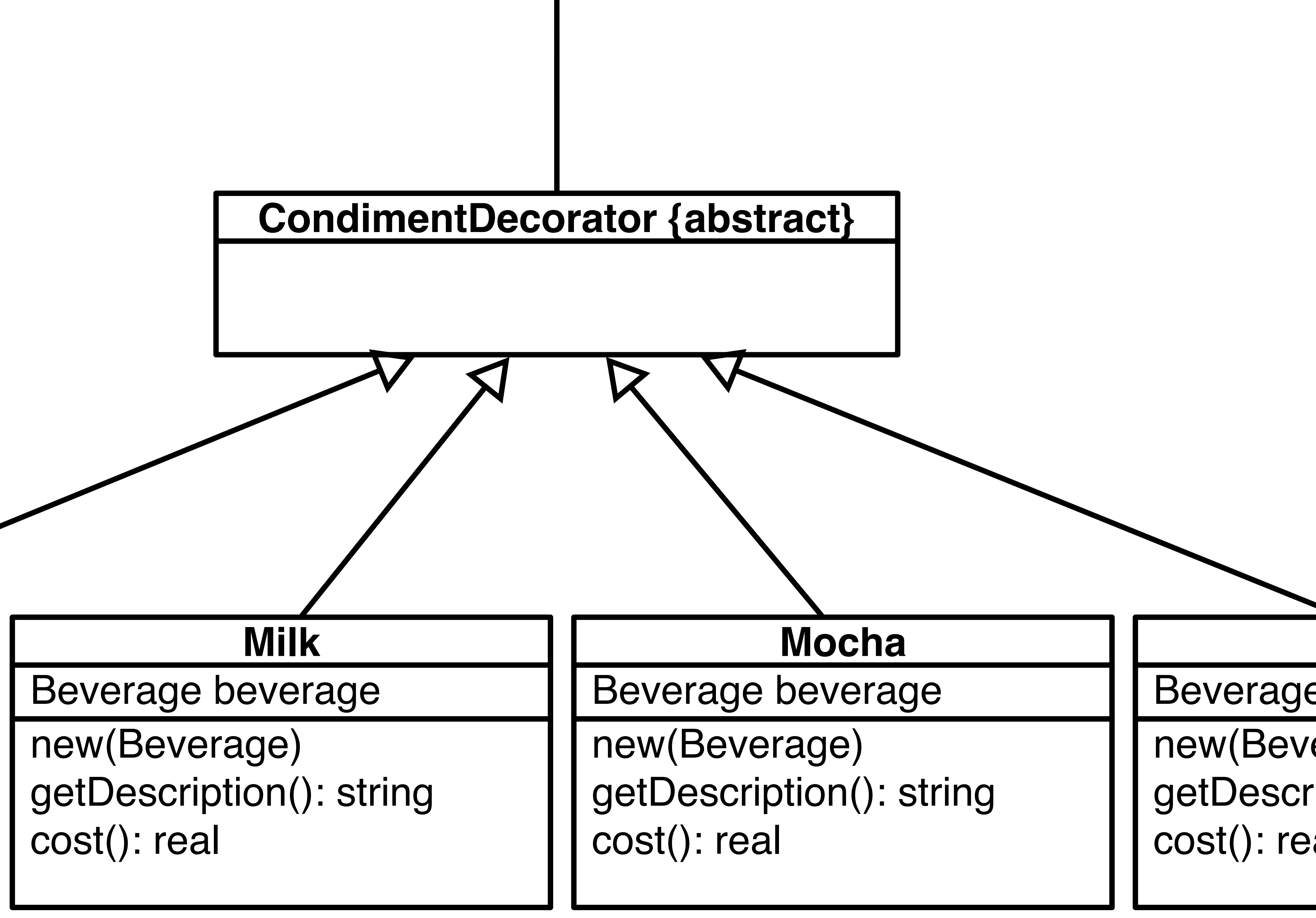
Decorator Pattern Example Starbuzz Coffee

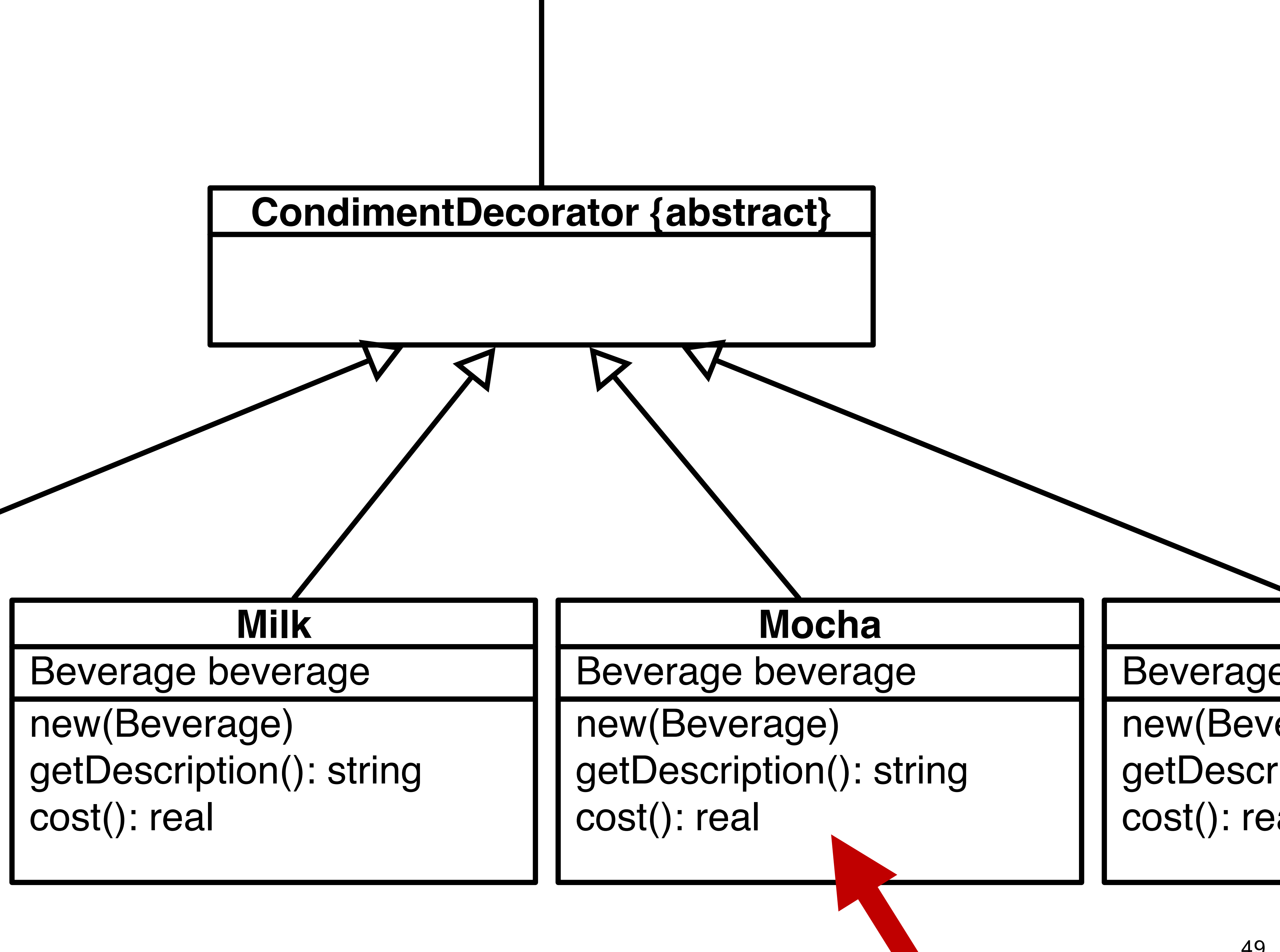
- Coffee type like: HouseBlend or DarkRoast
- Has Condiments like: Mocha or Whip
- Each Coffee type has a price and each Condiment has a price
- An order could be
  - *“Dark Roast with Double Mocha and Whip”*
  - Give the string and price for this order



starbuzz
Beverage {abstract}
description: string
getDescription(): string
cost(): real
Decaf
new()
cost(): real
DarkRoast
new()
cost(): real
HouseBlend
new()
cost(): real
CondimentDecorator {abstract}
Espresso
new()
cost(): real
Whip
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real
Milk
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real
Mocha
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real
Soy
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real


Beverage {abstract}
description: string
getDescription(): string
cost(): real
DarkRoast
new()
cost(): real
CondimentDecorator {abstract}
Espresso
new()
cost(): real


CondimentDecorator {abstract}
Milk
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real
Mocha
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real


CondimentDecorator {abstract}
Milk
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real
Mocha
Beverage beverage
new(Beverage)
getDescription(): string
cost(): real

```
class Mocha extends CondimentDecorator;

   Beverage beverage;

   function new(Beverage beverage);
      this.beverage = beverage;
   endfunction

   virtual function string getDescription();
      return {beverage.getDescription(), ", Mocha"};
   endfunction

   virtual function real cost();
      return (0.20 + beverage.cost());
   endfunction

endclass
```

```
class Mocha extends CondimentDecorator;

   Beverage beverage;

   function new(Beverage beverage);
      this.beverage = beverage;
   endfunction

   virtual function string getDescription();
      return {beverage.getDescription(), ", Mocha"};
   endfunction

   virtual function real cost();
      return (0.20 + beverage.cost());
   endfunction

endclass
```

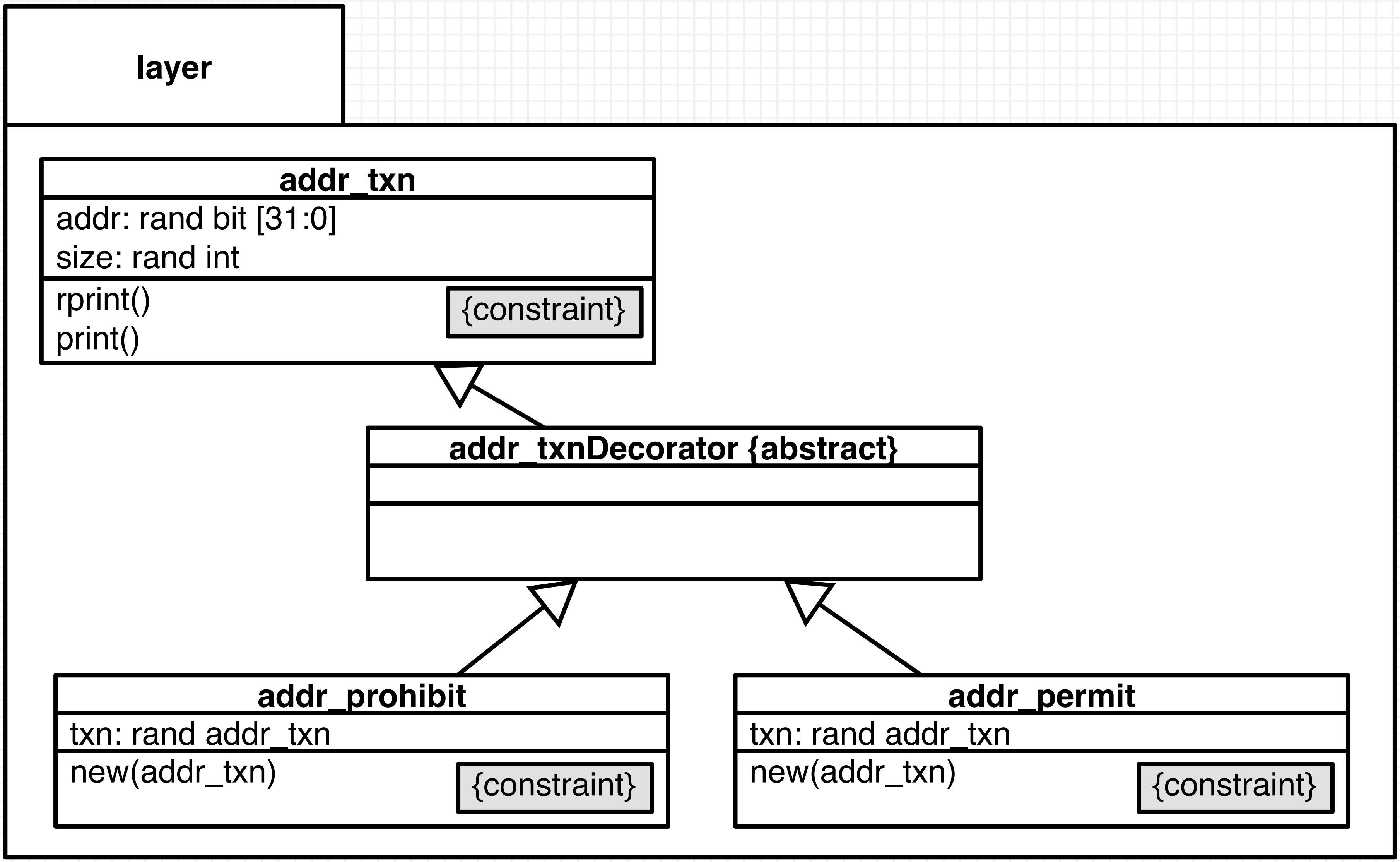
```
module top;
   import starbuzz::*;

   Beverage beverage;
   string str;

   initial begin
      beverage  = Darkroast::new;
      beverage  = Mocha::new(beverage);
      beverage  = Mocha::new(beverage);
      beverage  = Whip::new(beverage);
      str.realtoa(beverage.cost());
      $display({beverage.getDescription(), " $", str});
```

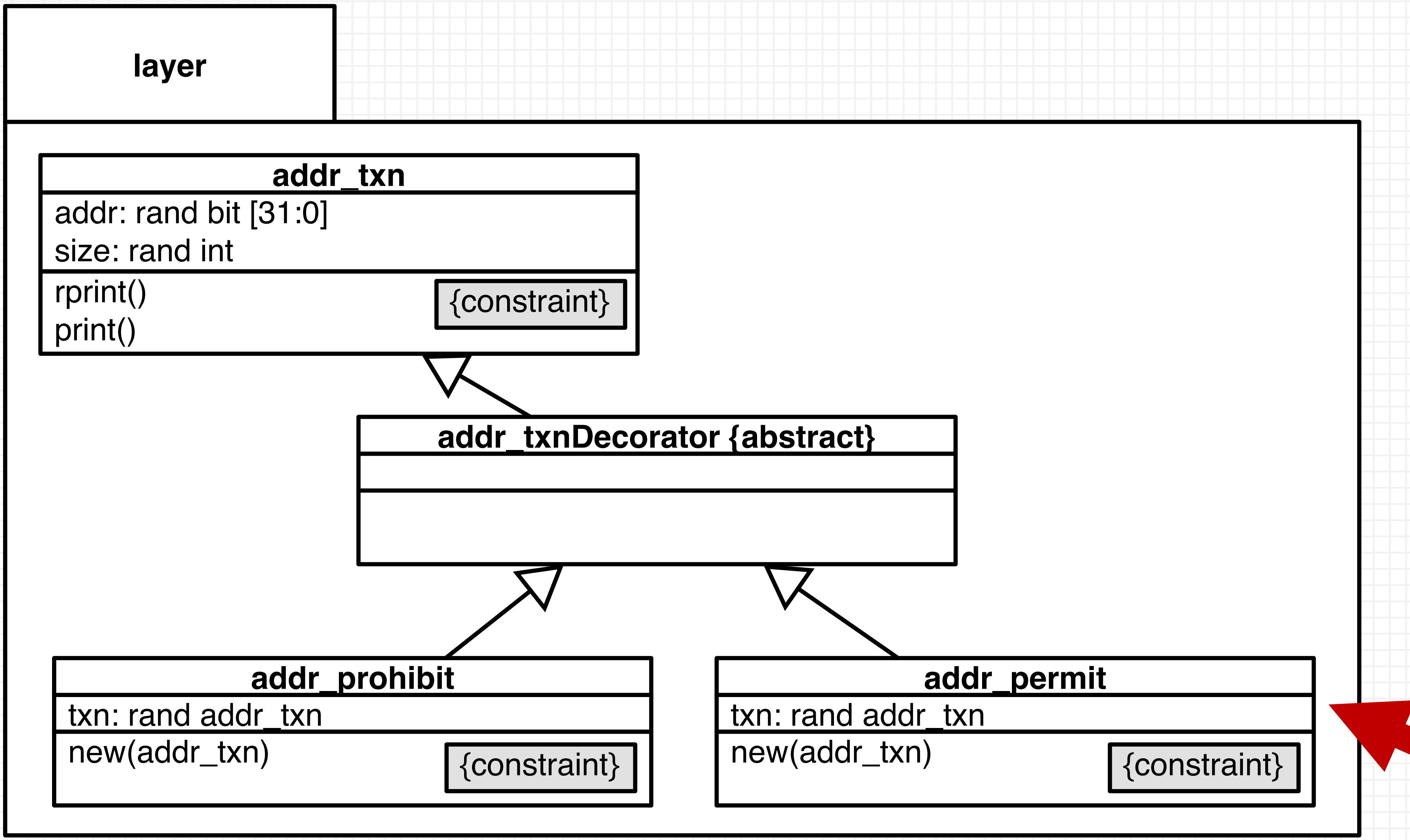
New SV
1800-2012

"Dark Roast Mocha Mocha Whip $1.65"

Decorator Pattern Application

- “SystemVerilog Constraint Layering via Reusable Randomization Policy Classes”
  John Dickol, DVCon 2015
- Dynamically applying multiple combinations of SystemVerilog constraints at runtime


layer
addr_txn
addr: rand bit [31:0]
size: rand int
rprint()
print()
{constraint}
addr_txnDecorator {abstract}
addr_prohibit
txn: rand addr_txn
new(addr_txn)
{constraint}
addr_permit
txn: rand addr_txn
new(addr_txn)
{constraint}


layer
addr_txn
addr: rand bit [31:0]
size: rand int
rprint()
print()
{constraint}
addr_txnDecorator {abstract}
addr_prohibit
txn: rand addr_txn
new(addr_txn)
{constraint}
addr_permit
txn: rand addr_txn
new(addr_txn)
{constraint}

```
class addr_permit extends addr_txnDecorator;
   rand addr_txn txn;

   function new(addr_txn txn);
      this.txn = txn;
   endfunction

   constraint c_addr_permit {
      addr inside {['h00000000 : 'h0000FFFF - txn.size]} ||
      addr inside {['h10000000 : 'h1FFFFFFF - txn.size]};

      txn.addr == addr;
      txn.size == size;
   }

endclass
```

```
class addr_permit extends addr_txnDecorator;
  rand addr_txn txn;

  function new(addr_txn txn);
    this.txn = txn;
  endfunction

  constraint c_addr_permit {
    addr inside {['h00000000 : 'h0000FFFF - txn.size]} ||
    addr inside {['h10000000 : 'h1FFFFFFF - txn.size]};

    txn.addr == addr;
    txn.size == size;
  }

endclass
```

```
module top;

  layer::addr_txn txn;

  initial begin
    txn = new;
    txn = layer::addr_prohibit::new(txn);
    txn = layer::addr_permit::new(txn);
    txn.rprint();
```

Decorator Pattern
Layered Constraint

Awesome Let's Use It!

- Software Design Pattern ideas are widely accepted and refined over decades
- The examples ported to SystemVerilog have a high fidelity with their Java sources
  - Meaning SystemVerilog has feature parity to meet these examples without compromises with SV 2012
- Simulator Vendor Compatibility

SV 2012 Simulator Compatibility

Simulator	local constructor	keyword "implements"	keyword "interface class"	typed constructor calls
A	Y	Y	Y	Y
B	Y	Y	Y	N
C	Y	N	N	N

New SV 1800-2012

New SV 1800-2012

New SV 1800-2012

Design Pattern Compatibility per Simulator

Simulator	Decorator	Singleton	Observer	Strategy	State
A	P	P	P	P	P
B	S	P	P	P	P
C	S	P	N	N	N
P (Preferred Implementation) → S (Satisfactory Implementation) → N (Not Directly Supported)					

Outline

- Design Patterns
  - Approach
- Composition versus Inheritance
- Strategy Pattern
  - Class Interface and Implements
  - Application to Evolving Packet Class
- Decorator Pattern
  - Typed Constructor Calls
  - Application to Layered Constraints

Opportunities with Design Patterns

- Excited to see how these new language features give an alternative to inheritance
- Excited to see more use of the Decorator Pattern as it is one of the most common
  - constraints are just the beginning
- Vendor support will come
  - but please tell them you want it!

Questions

Please Vote at
http://vote.dvcon.org

Shy Audience Questions

1. Why use an "interface class" when you could use an "virtual class" and put in your own "pure virtual functions"?
2. Show me the horror that is the Decorator Pattern without "typed constructor calls"
3. What can I do to improve the adoption of these SystemVerilog features in the simulators?
4. Is the paper better than your presentation – cause… you know?

Shy Audience Questions 2

1. Why do you sign your @ieee.org email address on all your papers?
2. What pattern if you had more time would you like to share next? Well, other than the Packet Protocol protocol one you said you skipped.

Backup Slides

UVM Policy Not Exactly Strategy Pattern

uvm_default_printer

```
uvm_printer uvm_default_printer = uvm_default_table_printer
```

The default printer policy. Used when calls to uvm_object::print or uvm_object::sprint do not specify a printer policy.

The default printer may be set to any legal uvm_printer derived type, including the global line, tree, and table printers described above.

```
module top;
   import starbuzz::*;

   Beverage beverage;
   string str;

   initial begin
      beverage  = Darkroast::new;
      beverage  = Mocha::new(beverage);
      beverage  = Mocha::new(beverage);
      beverage  = Whip::new(beverage);
      str.realtoa(beverage.cost());
      $display({beverage.getDescription(), " $", str});
```

Decorator Pattern **With** Typed Constructor Calls

New SV 1800-2012

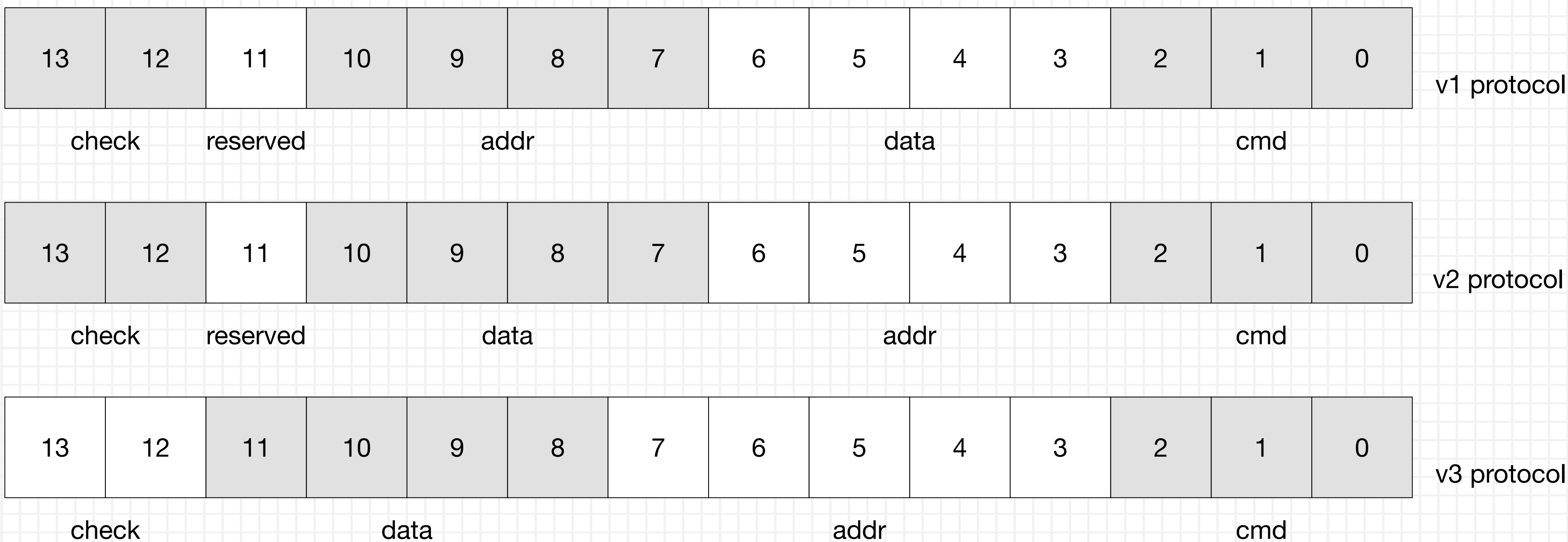
"Dark Roast Mocha Mocha Whip $1.65"

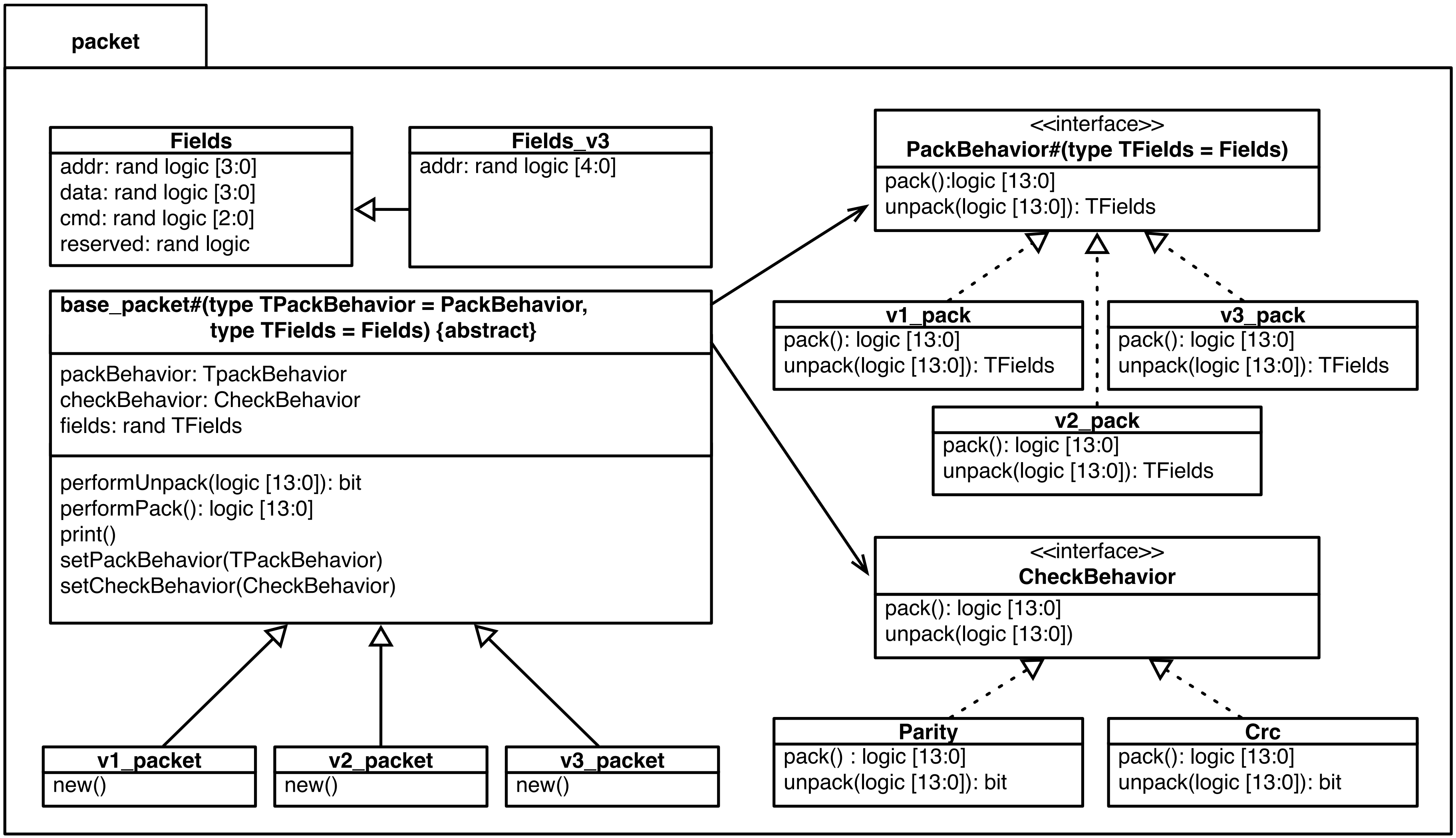
Decorator Pattern **Without** Typed Constructor Calls

```
module top;
   import starbuzz::*;

   Beverage beverage;
   DarkRoast darkroast;
   Mocha mocha;
   Whip whip;
   string str;

   initial begin
      darkroast  = new;
      beverage   = new darkroast;
      mocha      = new(beverage);
      beverage   = new mocha;
      mocha      = new(beverage);
      beverage   = new mocha;
      whip       = new(beverage);
      beverage   = new whip;
      str.realtoa(beverage.cost());
      $display({beverage.getDescription(), " $", str});
```

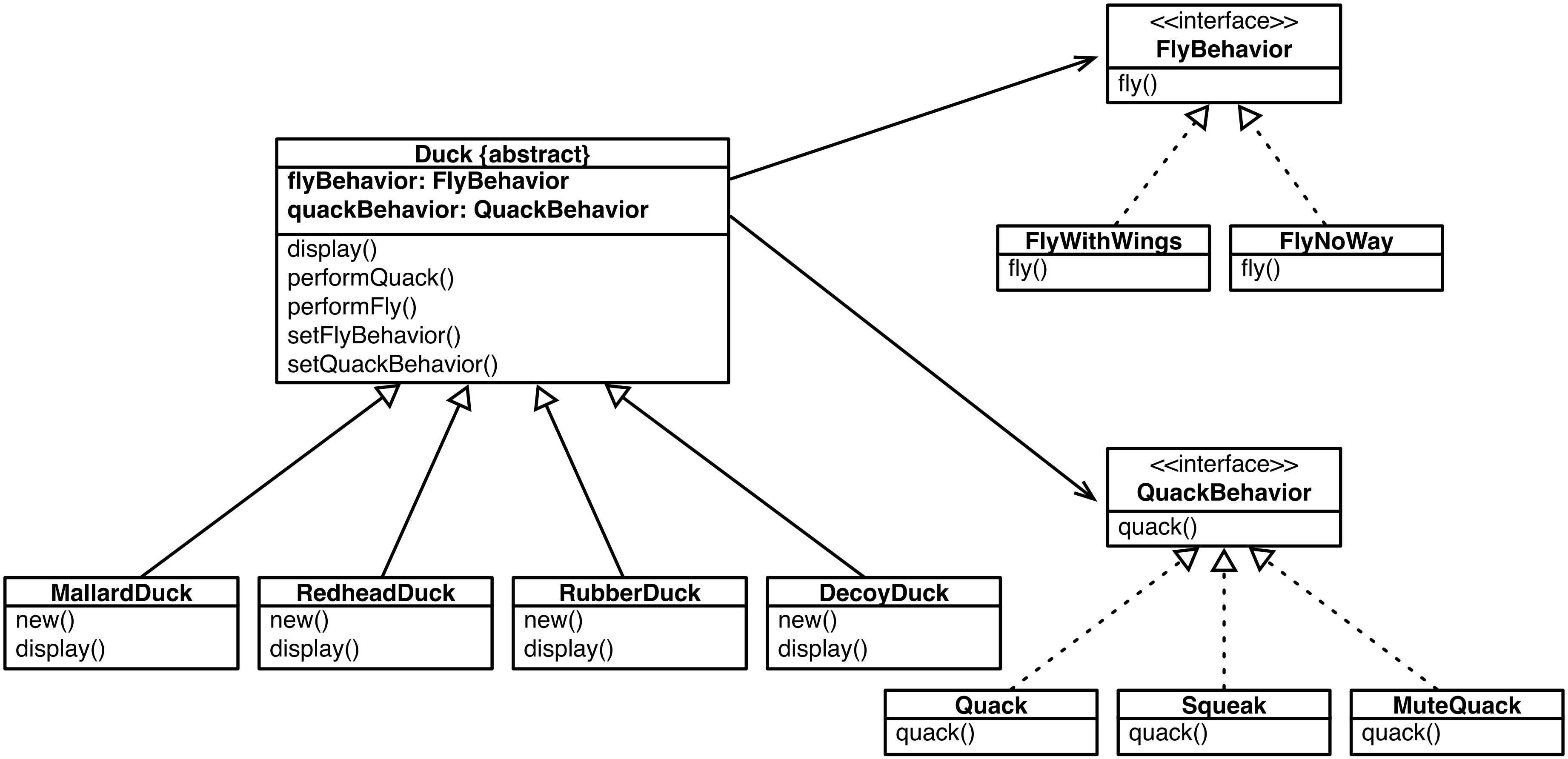
Strategy Pattern

- Removed from Presentation for Time Issues

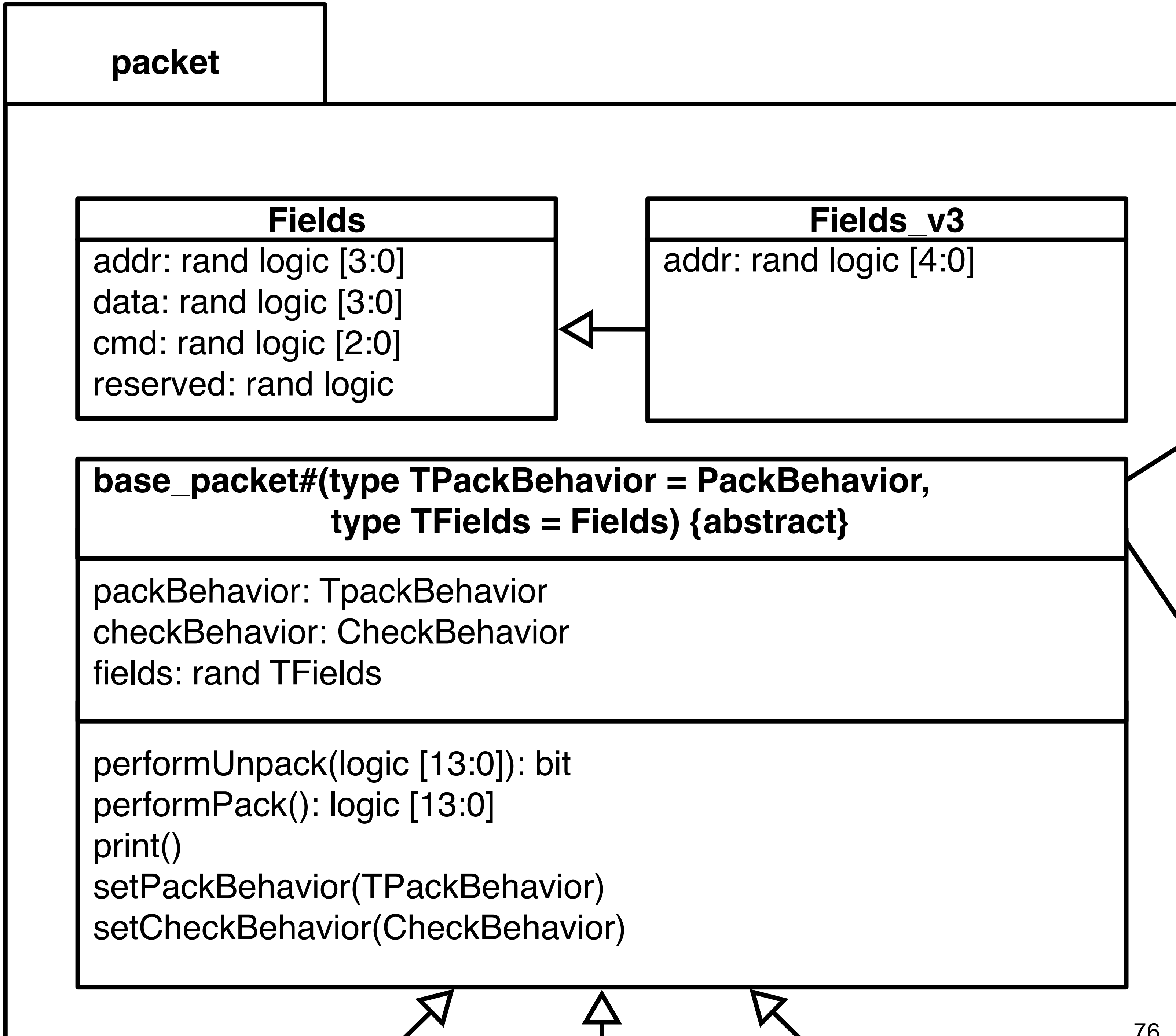

13 12 11 10 9 8 7 6 5 4 3 2 1 0
v1 protocol
check reserved addr data cmd
13 12 11 10 9 8 7 6 5 4 3 2 1 0
v2 protocol
check reserved data addr cmd
13 12 11 10 9 8 7 6 5 4 3 2 1 0
v3 protocol
check data addr cmd

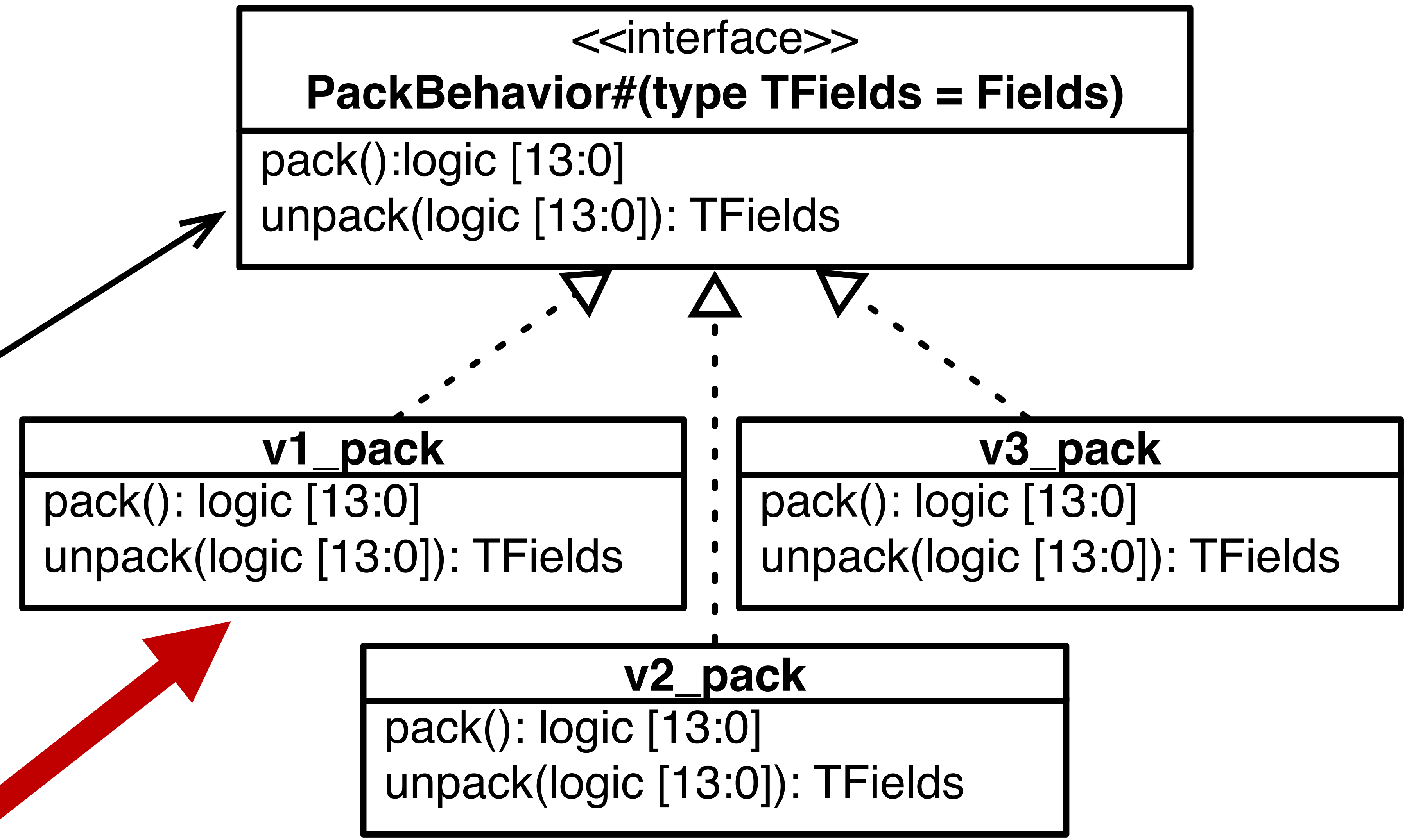


packet
Fields
addr: rand logic [3:0]
data: rand logic [3:0]
cmd: rand logic [2:0]
reserved: rand logic
Fields_v3
addr: rand logic [4:0]
<<interface>>
PackBehavior#(type TFields = Fields)
pack():logic [13:0]
unpack(logic [13:0]): TFields
base_packet#(type TPackBehavior = PackBehavior,
type TFields = Fields) {abstract}
packBehavior: TpackBehavior
checkBehavior: CheckBehavior
fields: rand TFields
performUnpack(logic [13:0]): bit
performPack(): logic [13:0]
print()
setPackBehavior(TPackBehavior)
setCheckBehavior(CheckBehavior)
v1_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields
v3_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields
v2_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields
<<interface>>
CheckBehavior
pack(): logic [13:0]
unpack(logic [13:0])
v1_packet
new()
v2_packet
new()
v3_packet
new()
Parity
pack() : logic [13:0]
unpack(logic [13:0]): bit
Crc
pack(): logic [13:0]
unpack(logic [13:0]): bit

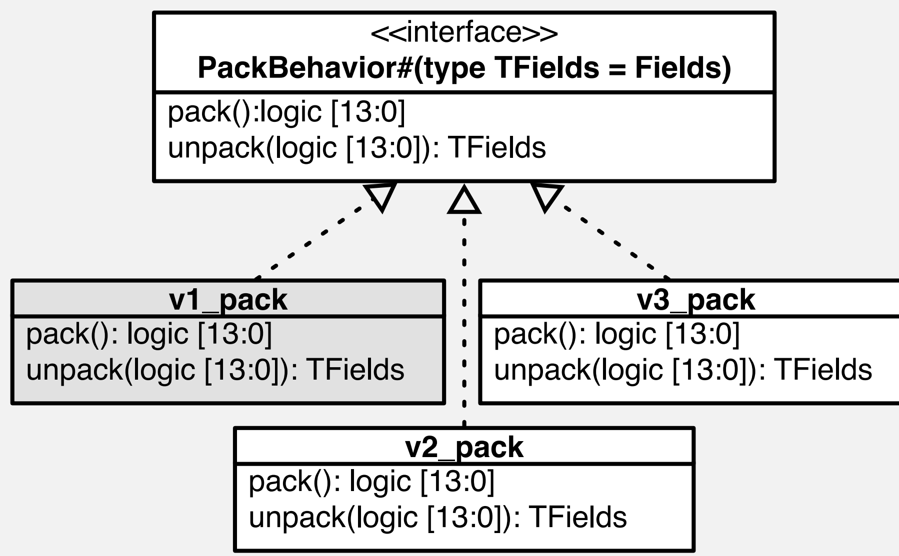


<<interface>>
FlyBehavior
fly()
Duck {abstract}
flyBehavior: FlyBehavior
quackBehavior: QuackBehavior
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
FlyWithWings
fly()
FlyNoWay
fly()
<<interface>>
QuackBehavior
quack()
MallardDuck
new()
display()
RedheadDuck
new()
display()
RubberDuck
new()
display()
DecoyDuck
new()
display()
Quack
quack()
Squeak
quack()
MuteQuack
quack()


packet
Fields
addr: rand logic [3:0]
data: rand logic [3:0]
cmd: rand logic [2:0]
reserved: rand logic
Fields_v3
addr: rand logic [4:0]
base_packet#(type TPackBehavior = PackBehavior,
type TFields = Fields) {abstract}
packBehavior: TpackBehavior
checkBehavior: CheckBehavior
fields: rand TFields
performUnpack(logic [13:0]): bit
performPack(): logic [13:0]
print()
setPackBehavior(TPackBehavior)
setCheckBehavior(CheckBehavior)


<<interface>>
PackBehavior#(type TFields = Fields)
pack():logic [13:0]
unpack(logic [13:0]): TFields
v1_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields
v3_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields
v2_pack
pack(): logic [13:0]
unpack(logic [13:0]): TFields

```
class v1_pack implements PackBehavior;
   virtual function Fields unpack(logic [13:0] raw);
      Fields fields     = new;
      fields.reserved  = raw[11];
      fields.addr      = raw[10:7];
      fields.data      = raw[6:3];
      fields.cmd       = raw[2:0];
      return fields;
   endfunction
```



```
class v1_packet extends base_packet;
   function new();
      v1_pack p = new();
      Crc c = new();

      setPackBehavior(p);
      setCheckBehavior(c);
   endfunction
endclass
```