Polymorphism and Classes

CS3081 Program Design and Development

Coupling and Classes : Is it Composition or Inheritance ?

"has-a" = Containment and Composition

Employee "has-a" name = member of class.
Employee "has-a" UserAccount = UserAccount object is member.

"is-a" = Inheritance

Part-Time Employee "is-a" specialization of Employee,
PartTime inherits from Employee.

McConnel Examples

- Liskov Principle *"Subclasses must be usable through the base class interface."*
  - Bank Accounts : Interest Bearing VS Interest Charging (p. 144)
- Overriding routines that do nothing.
  - ScratchlessTaillessMicelssMilklessCat (p. 146)

Inheritance and Composition : not as distinct as you think

```
class XClass {
private:
     int i;
public:
     XClass() : i(0) {}
     void subtract(int in) { i = i - in; }
     void add(int in) { i = i + in; }
     void display() { printf("i in X: %d \n", i); }
};
```

```
class YClassInh : public XClass {
private:
     int i;
public:
     YClassInh() : i(0) {}
     void subtract(int in) { i = i – 2*in; }
     void display() { printf("i in Y: %d \n", i); }
     void displayX() { XClass::display(); }
};
```

Inherited Class

```
int main() {
     YClassInh Y;
     Y.display();
     Y.displayX();

     Y.add(10);
     Y.display();
     Y.displayX();

     Y.subtract(10);
     Y.display();
     Y.displayX();

     Y.XClass::display();
```

```
> ./a.out
i in Y: 0
i in X: 0

i in Y: 0
i in X: 10

i in Y: -20
i in X: 10
```

```
class YClassPriv {
private:
     int i;
     XClass X;
public:
     YClass() : i(0) {}
     void change(int in) { i = i - in; }
     void display() { printf("i in Y: %d \n", i); }
     void changeX(int in) { X.change(in); }
     void displayX() { X.display(); }
};
```

Private Embedded Object

```
int main() {
     YClassPriv Y;
     Y.display();
     Y.displayX();
     Y.change(10);
     Y.changeX(10);
     Y.display();
     Y.displayX();
```

```
> ./a.out
i in Y: 0
i in X: 0
i in Y: -10
i in X: 10
```

Composition Versus Inheritance : Embedded Objects and Private Data

```
class objectClass {
private:
  int objectVar;
public:
  objectClass() : objectVar(10)
  objectClass(int a) : objectVar
  void print() {
    cout << "in objectClass. ";
    cout << "objectVar: " << obj
  }
};
```

```
class composedClass {
private:
  objectClass object;
public:
  void print() {
    cout << "in composedCl
   // cout << "objectVar "
   // cout << "objectVar "
    object.print();
  }
};
```

```
class derivedClass : public objectC
private:

public:
  void print() {
    cout << "in derivedClass. ";
    //cout << "objectVar " << objec
    //cout << "objectVar " << objec
    objectClass::print();
  }
};
```

What objects and data are available in the composed (aggregate) and derived class ?

```
cout << endl << "Embedded objects in both inherited and composed." << endl;
composedClass composedObject;
composedObject.print();
derivedClass derivedObject;
derivedObject.print();
```

```
Embedded objects in both inherited and composed.
in composedClass. in objectClass. objectVar: 10
in derivedClass. in objectClass. objectVar: 10
```

DerivedClass
Private:

Protected:

Public:

ObjectClass::
Private:
privateVar
Protected:
protectedVar
Public:
display()

DerivedClass
Private:

Protected:

Public:
display()

ObjectClass::
Private:
privateVar
Protected:
protectedVar
Public:
display()

DerivedClass
Private:

Protected:
protectedVar
Public:

ObjectClass::
Private:
privateVar
Protected:
protectedVar
Public:
display()

DerivedClass
Private:
privateVar
Protected:

Public:

ObjectClass::
Private:
privateVar
Protected:
protectedVar
Public:
display()

Referenced inside DerivedClass ...

privateVar
FAIL

protectedVar
ObjectClass::protectedVar

Display
ObjectClass::display()

privateVar
FAIL

protectedVar
ObjectClass::protectedVar

Display
DerivedClass::display()

Still call
ObjectClass::display()

privateVar

protectedVar
DerivedClass::protectedVar

Still have access to
ObjectClass::protectedVar

Display

privateVar
DerivedClass::privateVar

Still no access to the other

protectedVar

Display

Reusability : Key Motivation for Classes

- Reuse for efficiency.
- Reuse for safety.
- Reuse for easy modification.
- Reuse for simplification.

Functions are a reuse of code.
Embedded objects are a reuse of code.
Inheritance can be a reuse of design.

Polymorphism

Design Principle
Code should be closed to change,
yet open to extension.

Inheritance and Composition are forms of Reuse

Types of Inheritance:

- **Abstract Overridable Routine** : derived class inherits interface, not implementation
- **Overridable Routine** : derived class inherits interface and default implementation, can override implementation.
- **Non-Overridable Routine** : derived class inherits interface and default implementation. No override is allowed.

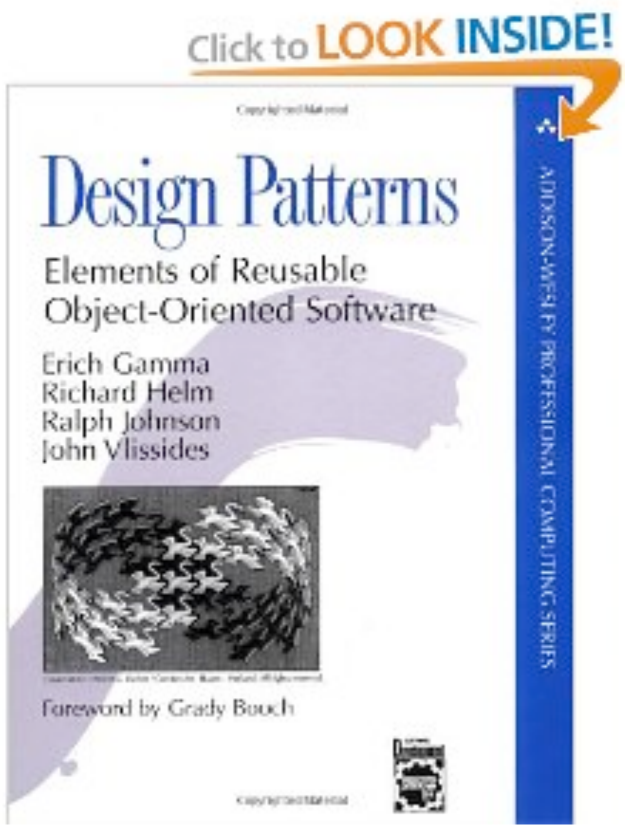
> You don't need to know these terms, just understand that there might be different agendas when combining classes.

"If you want to use an implementation but not its interface, use containment, not inheritance."

Design Patterns

- Originally introduced in *Design Patterns: Elements of Reusable Object-Oriented Software* by "Gang of Four (aka GOF)" Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

- *Head First Design Patterns* available on-line through UMN library.



Design Patterns

Categories of Design

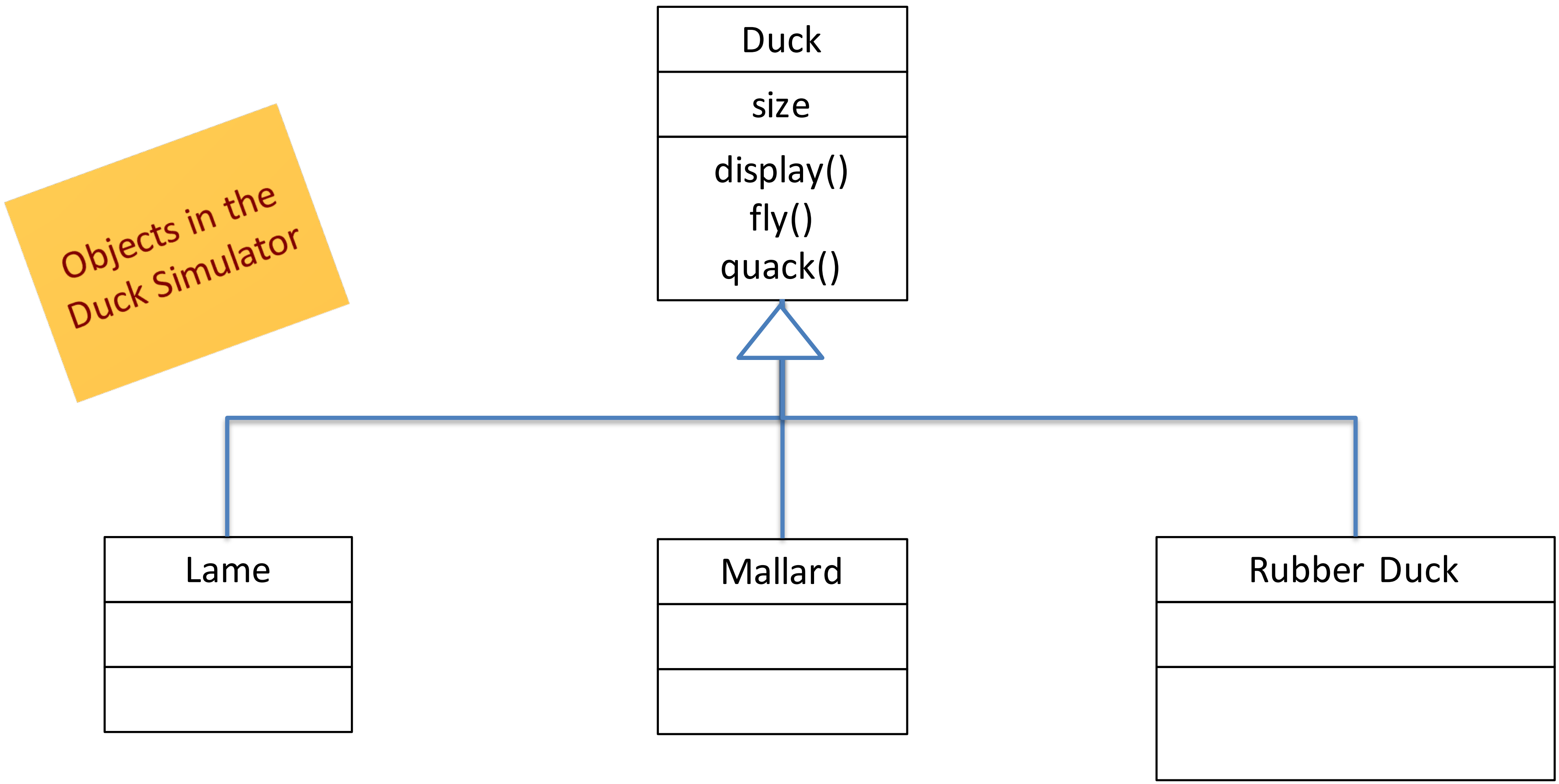
- Behavior (e.g. Strategy, Observer, ...)
- Compound (e.g. Model-View-Controller, ...)
- Creation (e.g. Factory, Singleton, ...)
- Structure (e.g. Adapter, Composite, ...)

Design Principle
Favor composition over inheritance.

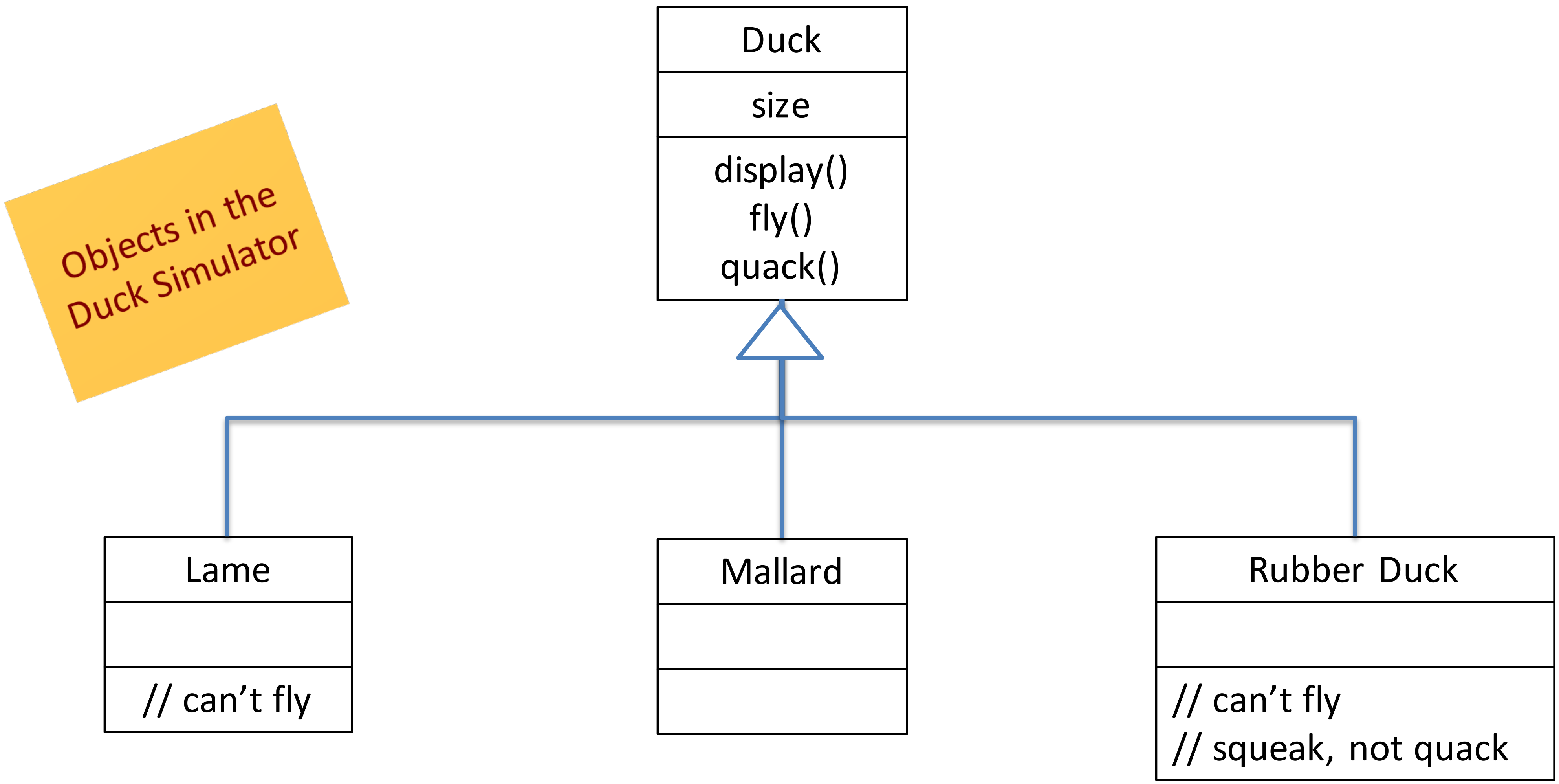
Design Principle
Identify aspects of your application that vary
and separate them from what stays the same.

Design Principle
Code should be closed to change,
yet open to extension.

Reusing Code Through Inheritance and Composition

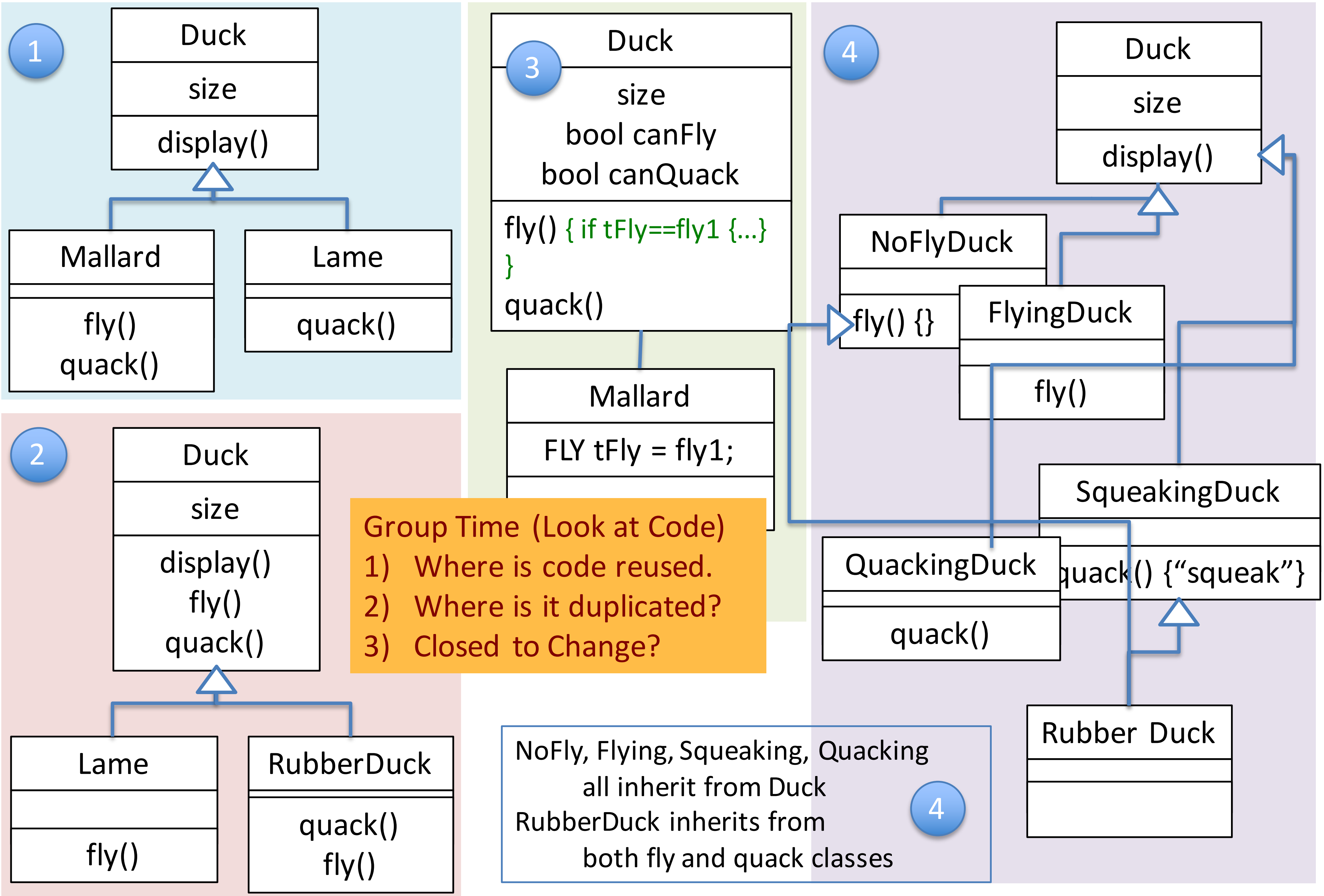


Reusing Code Through Inheritance and Composition



Problem : You don't want the subclass to inherit everything!

Reusing Code Through Inheritance and Composition

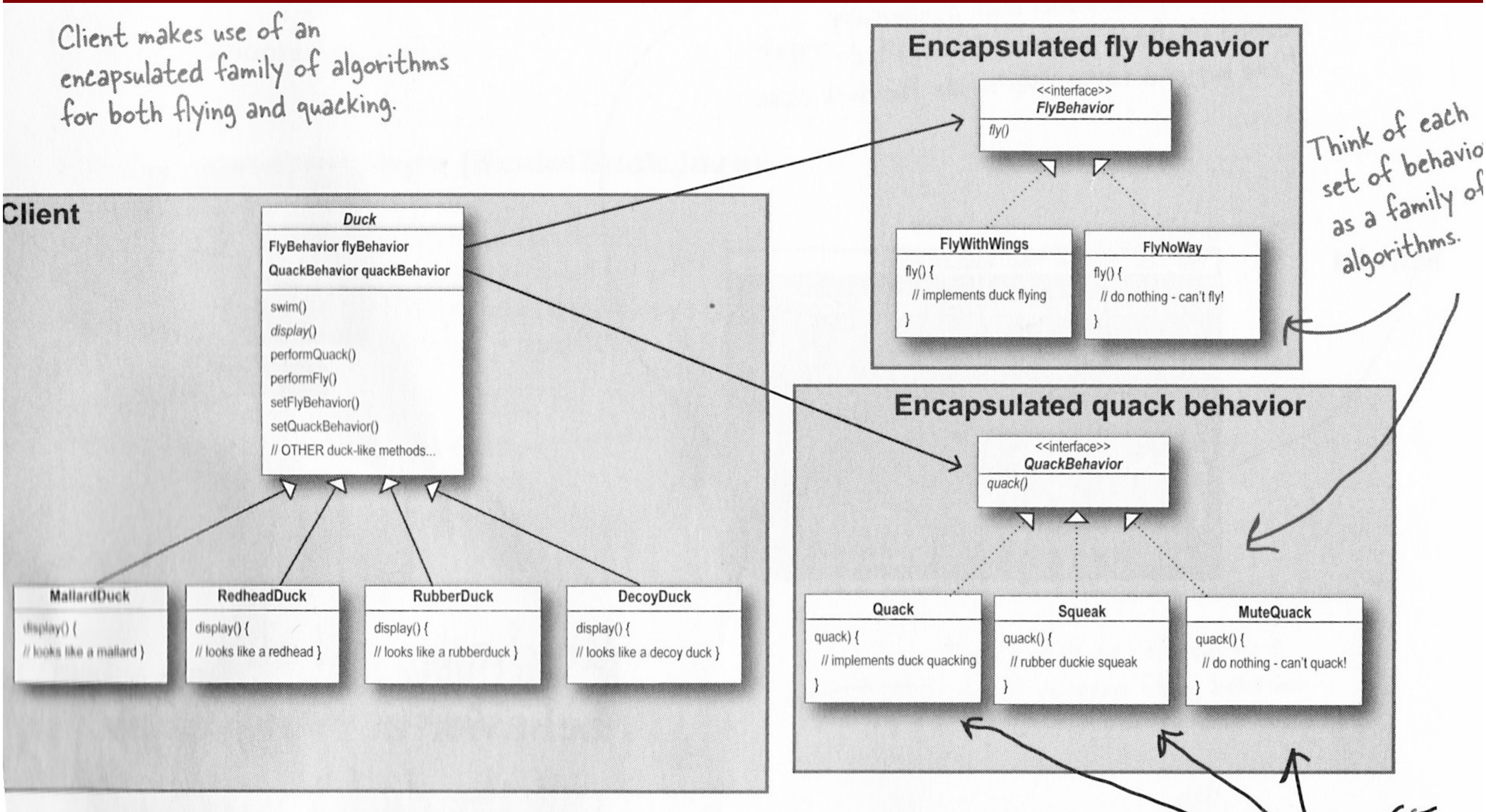


Group Time (Look at Code)

1) Where is code reused.
2) Where is it duplicated?
3) Closed to Change?

NoFly, Flying, Squeaking, Quacking all inherit from Duck
RubberDuck inherits from both fly and quack classes

A Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. (Design solution to the ScratchlessTaillessMiceIssMilklessCat.)



Design Principle
Identify aspects of your application that vary and separate them from what stays the same.

Design Principle
Favor composition over inheritance.

Duck Behaviors

- Add NoFly
- Add Rocket flying

Look at Code duckStrategy.cpp

1) Where is code reused.
2) Where is it duplicated?
3) Closed to Change?

- Review the output.
- What is wrong ??

```
Mary does this ...
I am a Mallard.
Generic Flying at 5 mph.
Generic Quack at 10 decibels

Ralph does this ...
I am a Rubber Duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels
```

Polymorphism

Polymorphism: generally defined as "the ability to create a variable, a function, or an object that has more than one form." The result is that you get different behavior (i.e. different pieces of code are executed) depending on the type of object or objects that are being acted upon.

- **Operator Overloading**: One operator can be applied to different types.
- **Method Overriding (Ad-hoc polymorphism)**: Derived class redefining base class method.
- **Method Overloading (Ad-hoc polymorphism)**: Multiple function definitions with different parameter lists.
- **Subtype Polymorphism**: Upcasting – derived class object can be used in place of base class object.
- **Parametric Polymorphism**: Templates – one function with same behavior across multiple types. (Stack of ints, strings, ClassA, ...)

cite: Wikipedia and http://www.catonmat.net/blog/cpp-polymorphism/

Achieving Polymorphism

```
class Instrument {
public:
  // This is to demonstrate what function is being called
  void play(note) const {
    cout << "Instrument::play" << endl;
  }
};
```

virtual

```
class Wind : public Instrument {
public:
  // This is to demonstrate what function is being called
  void play(note) const {
    cout << "Wind::play" << endl;
  }
};
```

```
void tune(Instrument i) {
  i.play(middleC);
}
```

& (or *)

```
int main() {
  Wind flute;

  // What do you expect to be the output of this call?
  tune(flute);
}
```

(or * with new)

Early Binding: a call to a class method is bound at compile-time.

virtual

Late Binding (or dynamic): a call to a class method is bound at runtime.

Polymorphic Ducks

```
class FlyBehavior {
private:
  double milesPerHour;
public:
  FlyBehavior() : milesPerHo
  virtual void fly() { cout
};

class FlyWithWings : public
private:
  double milesPerHour;
public:
  FlyWithWings() : milesPerH
  void fly() { cout << "Fly
};
```

```
Donald does this ...
I am a duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels

Mary does this ...
I am a Mallard.
Fly at speed of 5 mph.
Quack at 10 decibels

Ralph does this ...
I am a Rubber Duck.
Cannot fly at any speed.
Squeak at 10 decibels.
```

```
oes this ...
 Mallard.
c Flying at 5 mph.
c Quack at 10 decibels

does this ...
 Rubber Duck.
```

Strive for This.

What needs to change?

```
class Mallard : public Duck
private:
  FlyBehavior* flyBehavior;
  QuackBehavior* quackBehavi
public:
  Mallard();
  void display() { cout << "I am a Mallard." << en
  void quack() { quackBehavior->quack(); }
  void fly() { flyBehavior->fly(); }
};

Mallard::Mallard() {
  flyBehavior = new FlyWithWings;
  quackBehavior = new Quack;
}
```

```
es this ... " << endl;

cout << endl << "Mary does this ... " << endl;
mary.display();
mary.fly();
mary.quack();

cout << endl << "Ralph does this ... " << endl;
ralph.display();
ralph.fly();
ralph.quack();
```

Ducks, All in a Row

Strive for This.
What needs to change?

```
Duck ducks[3];
ducks[0] = Duck();
ducks[1] = Mallard();
ducks[2] = RubberDuck();

for (int i = 0; i < 3; i++) {
    cout << endl << "Duck[" << i << "]" << endl;
    ducks[i].display();
    ducks[i].fly();
    ducks[i].quack();
  }
```

```
Duck[0]
I am a duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels

Duck[1]
I am a duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels

Duck[2]
I am a duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels
```

```
Duck[0]
I am a duck.
Generic Flying at 5 mph.
Generic Quack at 10 decibels

Duck[1]
I am a Mallard.
Fly at speed of 5 mph.
Quack at 10 decibels

Duck[2]
I am a Rubber Duck.
Cannot fly at any speed.
Squeak at 10 decibels.
```

Dynamic Behavior!!

```
// I smashed into a window and broke my wing!
FlyBehavior* fb = new NoFly;
ducks[0]->setFly( fb );
cout << "Broken wing means ";
ducks[0]->fly();
```

```
Broken wing means Cannot fly at any speed.
```

Design Principles and Polymorphism

```
// to an implementation
Dog d = new Dog();
d.bark();

// to an interface
Animal animal = new Dog;
animal.speak();
```

Design Principle
Program to an interface, not an implementation.

Design Principle
Identify aspects of your application that vary and separate them from what stays the same.

Design Principle
Favor composition over inheritance.

Design Principle
Code should be closed to change, yet open to extension.

```
// New rocket jetpack for ducks!
FlyBehavior* fb2 = new FlyWithRocket;
Mallard* guineaPig = new Mallard;
cout << "No rocket, ";
guineaPig->fly();
guineaPig->setFly( fb2 );
cout << "With rocket, ";
guineaPig->fly();
```

```
No rocket, Fly at speed of 5 mph.
With rocket, Fly at speed of 500 mph.
```

```
class FlyWithRocket : public FlyBehavior {
private:
  double milesPerHour;
public:
  FlyWithRocket() : milesPerHour(MPH_DEFAULT*100) {}
  void fly() { cout << "Fly at speed of " << milesPerHour << " mph." << endl; }
};
```

Visitor Pattern

SITUATIONS:

- Need class member data, but only if of a certain subtype.
- Need to access or modify class member data, but application is different if subtype is different.

If subtypes are treated like parent class objects, type information is lost.

```
ObjectThatNeedsData ...
    for (int i; i<count; i++) {
        localData = SubtypeObject[i].getRelevantData()
        ... Do something with data ...
    }
```

```
Subtype1::getRelevantData() { return -1; }
Subtype2::getRelevantData() { return relevantData; }
```

To accept a visitor, means that you will pass yourself to the visitor. The visitor has a separate visit() for each subtype, therefore the compiler will match subtype to specific action.

```
//Putting It All Together

NeedsStuff1 ns1;

// define array of objects of various subtypes
PARENTCLASS objects[] ...

for (i=0;i<objCount;i++) {
        objects[i].accept( ns1 );
}
```

```
PARENTCLASS
----------------------
void accept(Visitor v) { v.visit(this); }
```

```
SUBCLASS1
----------------------
void accept(Visitor v)
```

```
SUBCLASS2
----------------------
void accept(Visitor v)
```

```
VISITOR CLASS
----------------------
void visit( SUBCLASS1)
void visit(SUBCLASS2)
```

```
NeedsStuff1
----------------------
void visit( SUBCLASS1) { // do nothing }
void visit( SUBCLASS2 { //get stuff }
```

```
NeedsStuff2
----------------------
void visit( SUBCLASS1) { // get stuff }
void visit( SUBCLASS2 { // do nothing }
```