Strategy Pattern

Prof. Jonathan Lee（李允中）
Department of CSIE
National Taiwan University

Design Aspect of Strategy

An algorithm or a family of algorithms

Outline

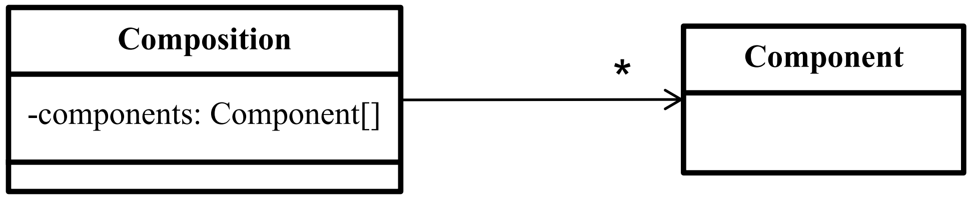
Text Composition Design (Strategy)

Prof. Jonathan Lee (李允中)

Department of Computer Science and Information Engineering
National Taiwan University

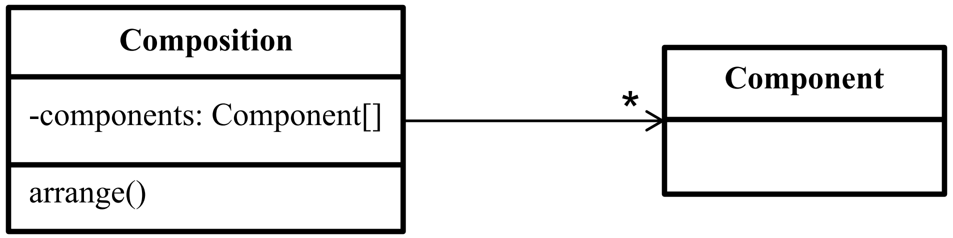
Requirements Statement$_1$

- The Composition class maintains a collection of Component instances, which represent text and graphical elements in a document.



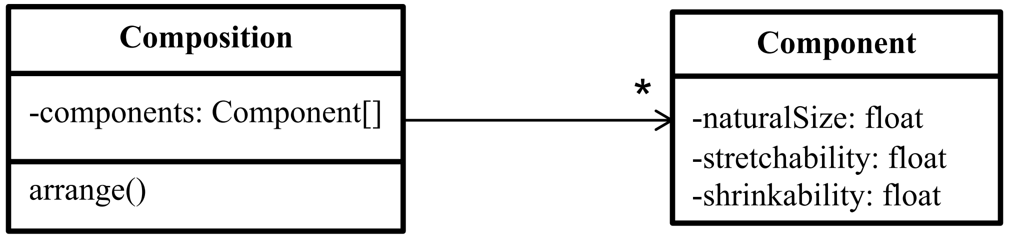
Requirements Statement$_2$

- A composition arranges component objects into lines using a linebreaking strategy.



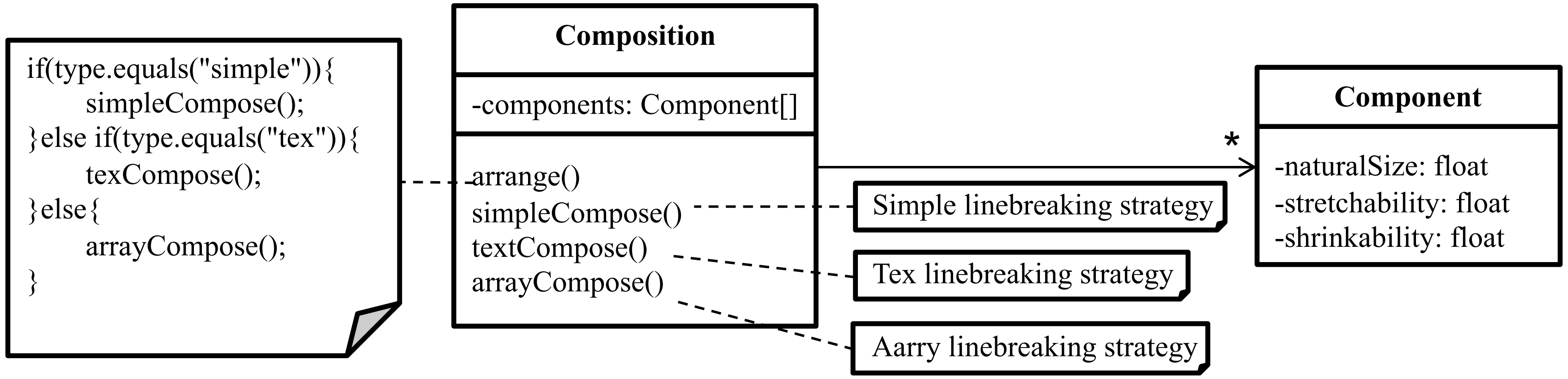
Requirements Statement$_3$

- Each component has an associated natural size, stretchability, and shrinkability.
- The stretchability defines how much the component can grow beyond its natural size; shrinkability is how much it can shrink.



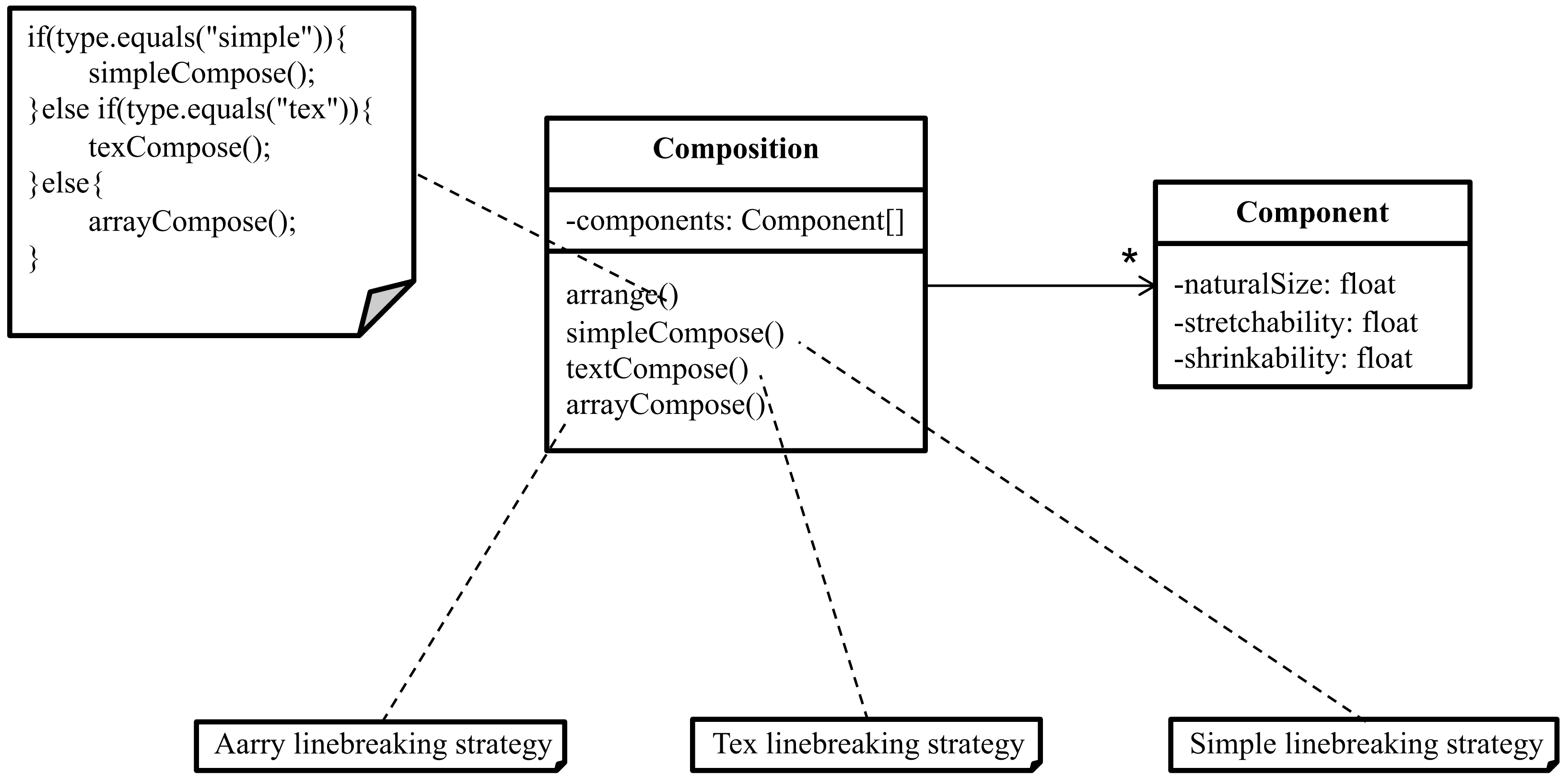
Requirements Statement$_4$

- When a new layout is required, the composition calls its compose method to determine where to place linebreaks.
- There are 3 different algorithms for breaking lines:
  - Simple Composition: A simple strategy that determines line breaks one at a time.
  - Tex Composition: This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
  - Array Composition: A strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.



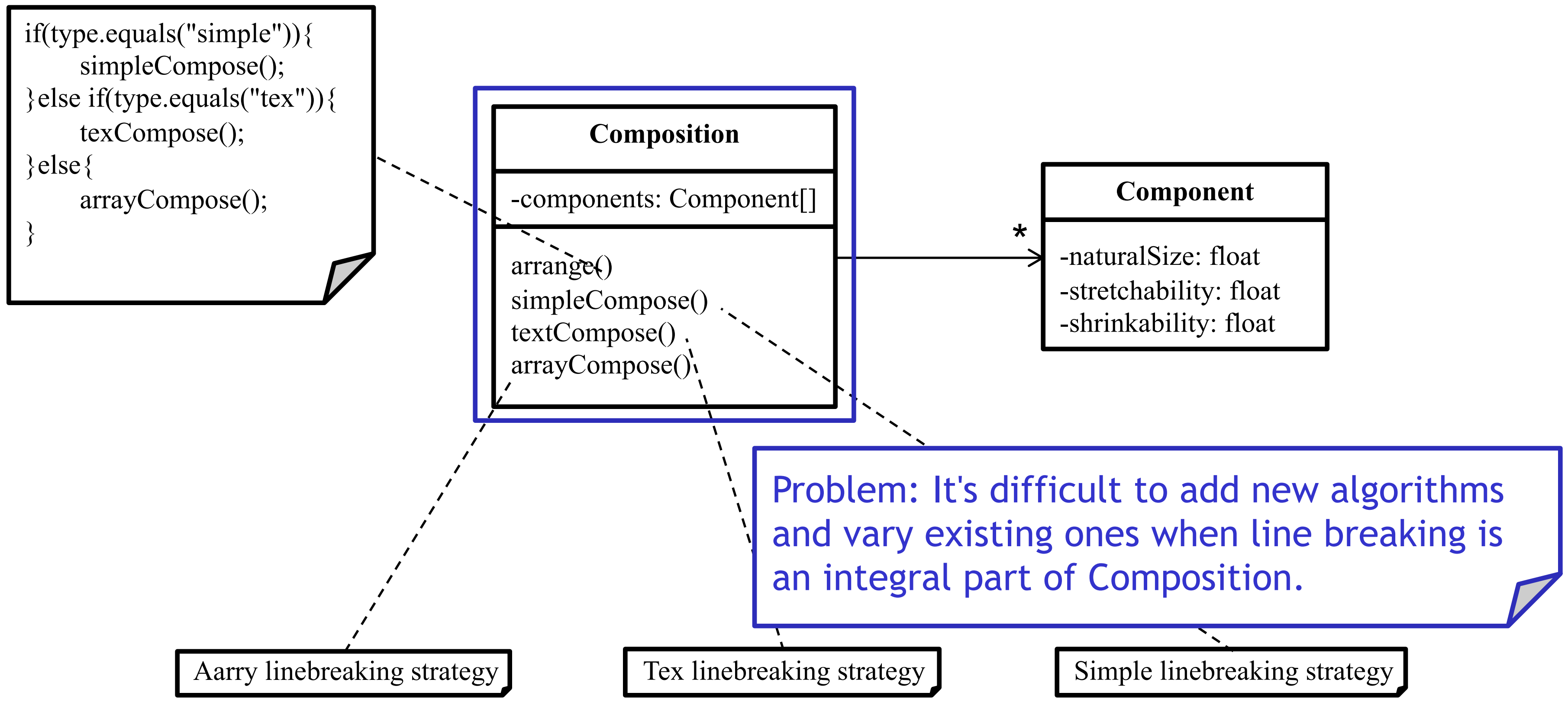
```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

Initial Design



```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

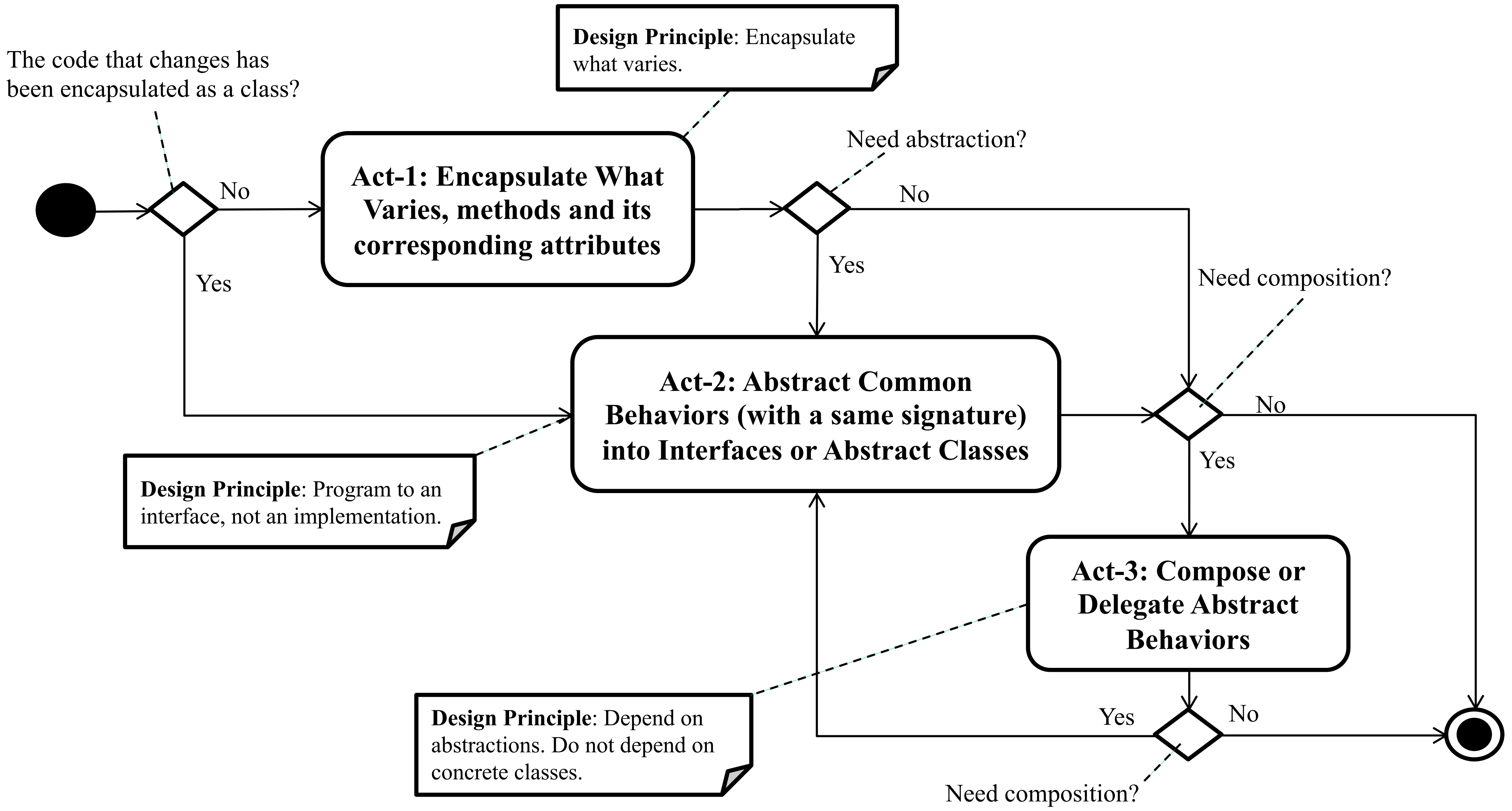
Problems with Initial Design



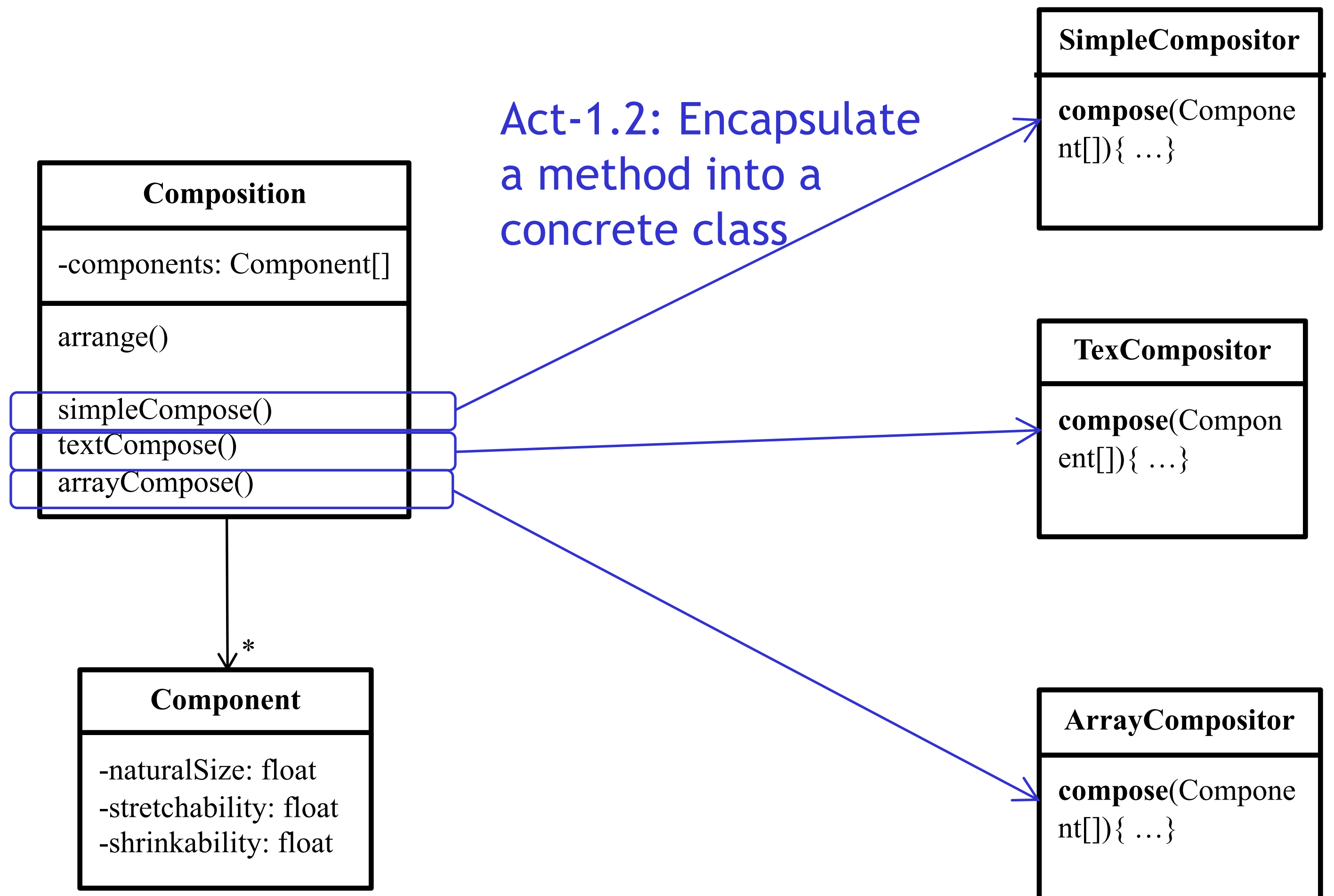
```
if(type.equals("simple")){
        simpleCompose();
}else if(type.equals("tex")){
        texCompose();
}else{
        arrayCompose();
}
```

Problem: It's difficult to add new algorithms and vary existing ones when line breaking is an integral part of Composition.

Design Process for Change

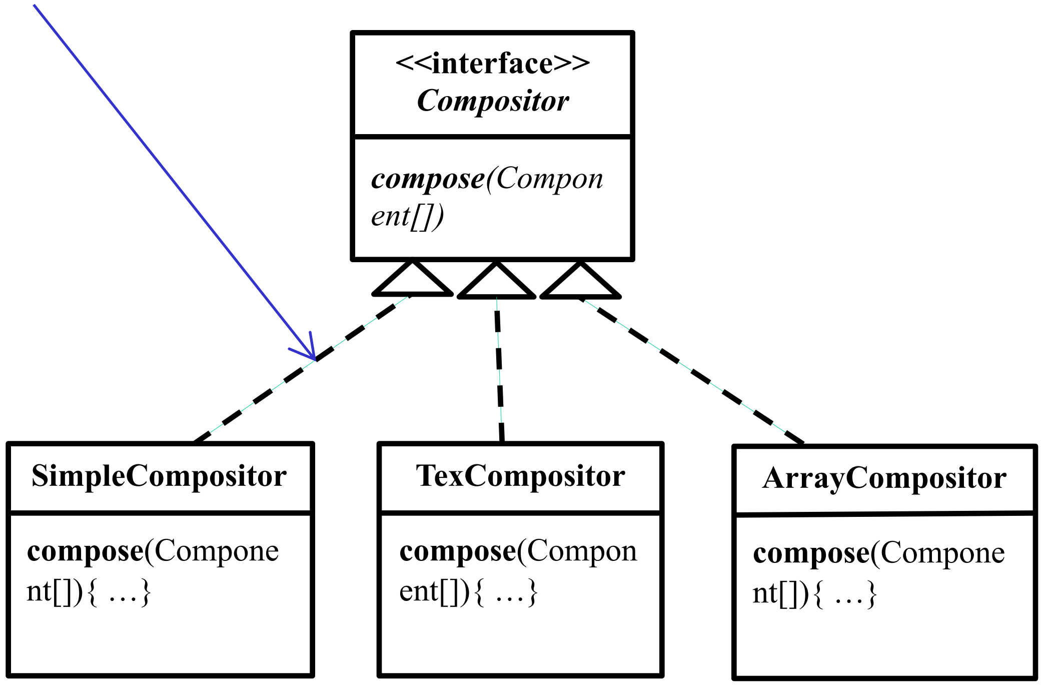

The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

Act-1: Encapsulate What Varies

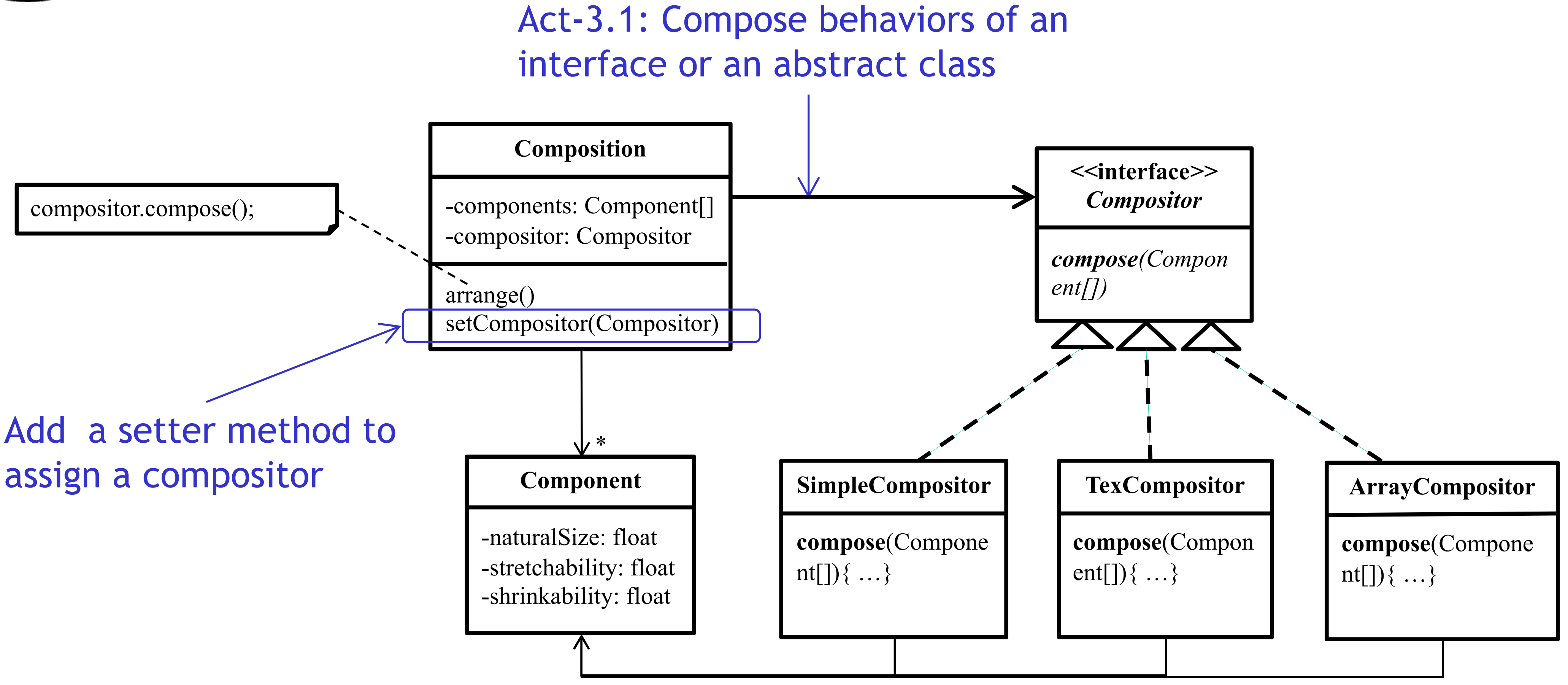


Act-2: Abstract Common Behaviors

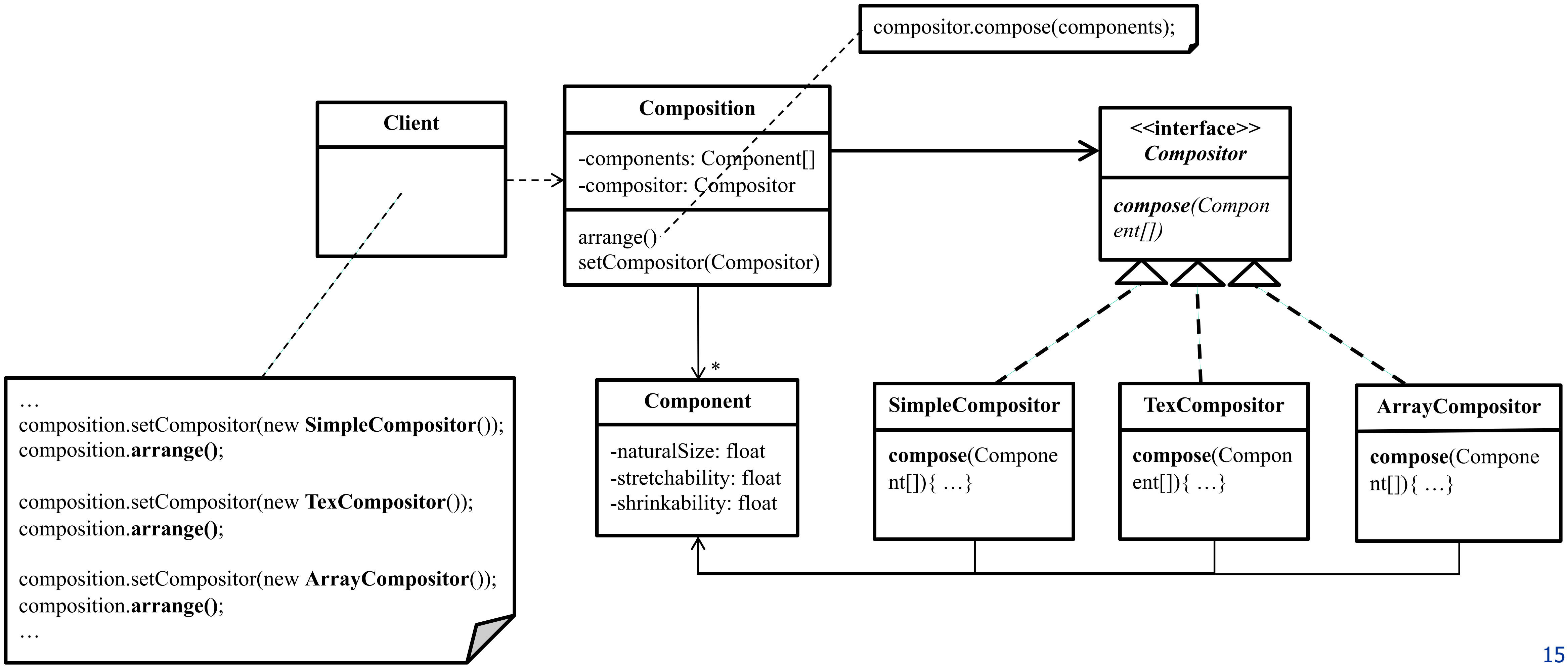
Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism



Act-3: Compose Abstract Behaviors



Refactored Design after Design Process



```
…
composition.setCompositor(new SimpleCompositor());
composition.arrange();

composition.setCompositor(new TexCompositor());
composition.arrange();

composition.setCompositor(new ArrayCompositor());
composition.arrange();
…
```

Constructor Injection

```
public class Client {
  public static void main(String[] argv) {
    Compositor cr = new SimpleCompositor();
    Composition com = new Composition(cr);
    …
    com.arrange();
  }
}
```

```
pubic interface Compositor {
  void compose(Component[] c);
}
public class  SimpleCompositor implements Compositor {
  public void compose(Component[] c)  {
     // do simple composition
  }
}
public class Composition {
  private Compositor cr;
  private Component[] components;
  public Composition(Compositor cr) {
    this.cr = cr;
  }
  public void arrange() {
    cr.compose(components);
  }
}
```

Setter Injection

```
public class Client {
  public static void main(String[] argv) {
    Composition com = new Composition();
    com.setCompositor(new SimpleCompositor();
    …
    com.arrange();
  }
}
```

```
pubic interface Compositor {
  void compose(Component[] c);
}
public class  SimpleCompositor implements Compositor {
  public void compose(Component[] c)  {
     // do simple composition
  }
}
public class Composition {
  private Compositor cr;
  private Component[] components;
  public void setCompositor(Compositor cr) {
    this.cr = cr;
  }
  public void arrange() {
    cr.compose(components);
  }
}
```

Method Injection

```
public class Client {
  public static void main(String[] argv) {
    Composition com = new Composition();
    com.arrange(new SimpleCompositor());
  }
}
```

```
pubic interface Compositor {
  void compose(Component[] c);
}
public class  SimpleCompositor implements Compositor {
  public void compose(Component[] c)  {
     // do simple composition
  }
}
public class TexCompositor implements Compositor {
  public void compose(Component[] c) {
    // do tex composistion;
  }
}
public class Composition {
  private Component[] components;
  public void arrange(Compositor cr) {
    cr.compose(components);
  }
}
```

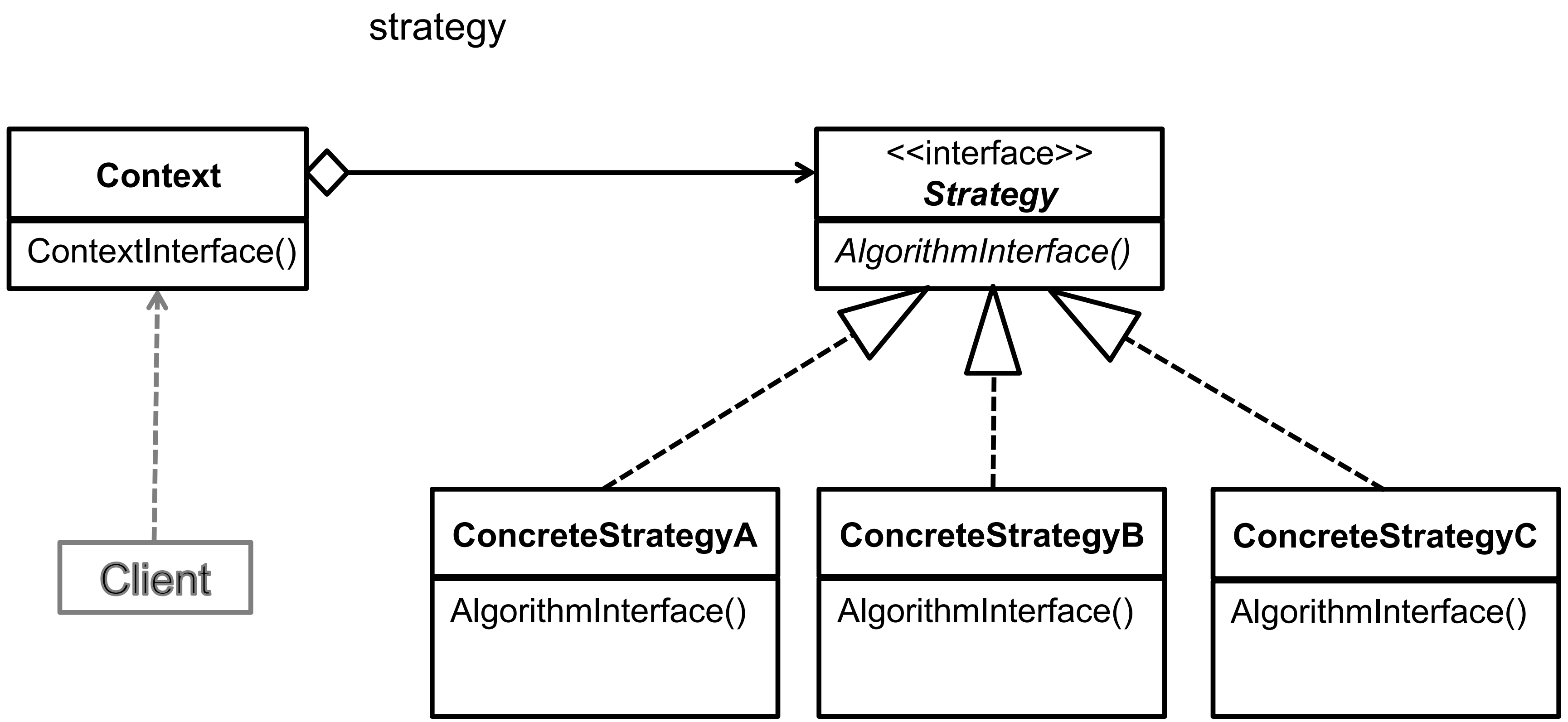
Recurrent Problems

- Multiple classes will be modified if new behaviors are to be added.
  - It's difficult to add new algorithms and vary from existing ones.
- All duplicate code will be modified if the behavior is to be changed.
  - Different algorithms may fit in different situations.

Intent

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

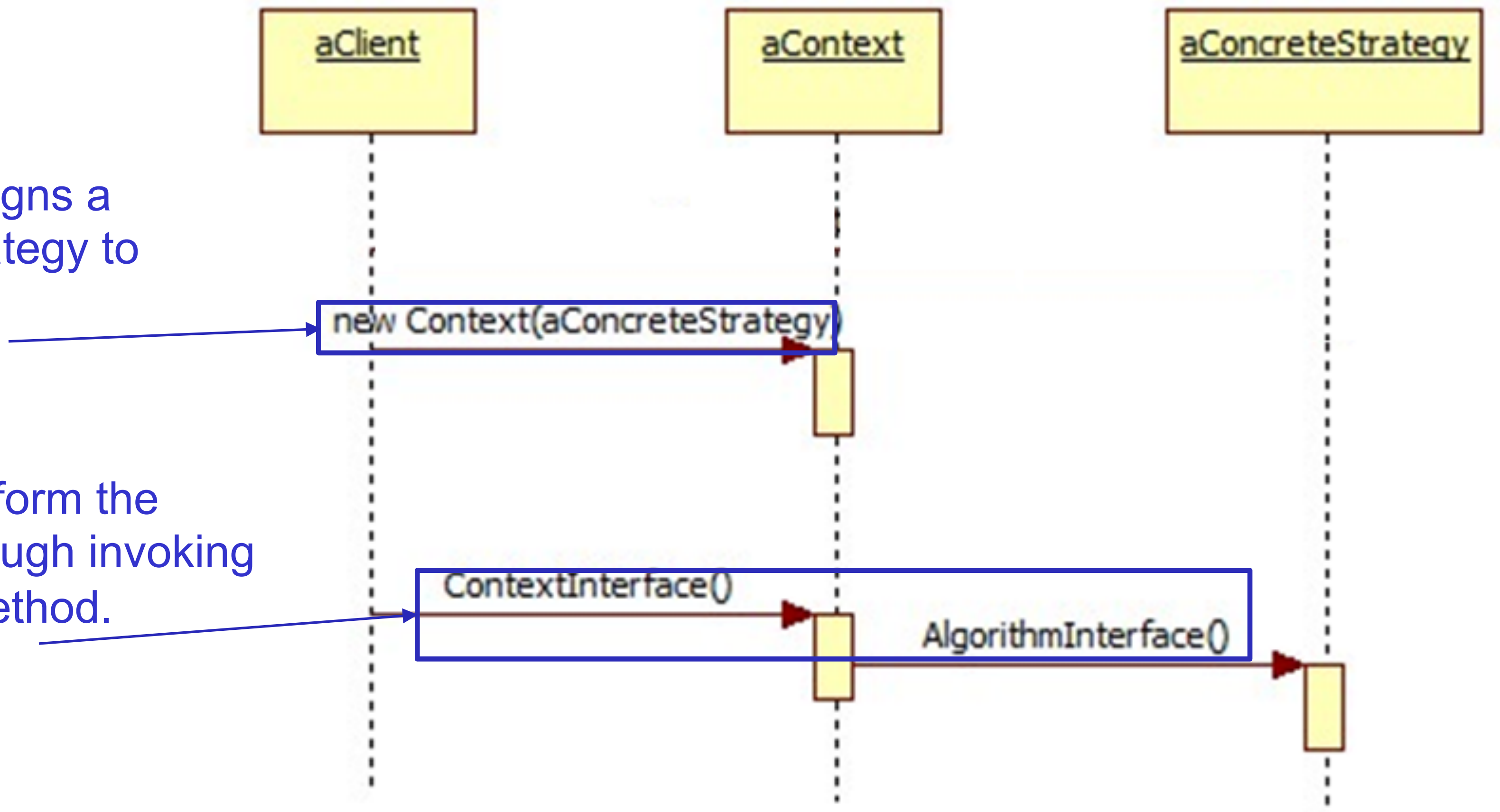
Strategy Pattern Structure$_1$



Strategy Pattern Structure$_2$

1. Client assigns a concrete strategy to Context.

2. Client perform the strategy through invoking Context's method.



Strategy Pattern Structure$_3$

	Instantiation	Use	Termination
Client	Other class except classes in the strategy pattern	Other class except classes in the strategy patterns	Other class except classes in the strategy pattern
Context	Other class or the client class	Client passes a ConcreteStrategy reference to this class and delegates request to it	Other class or the client class
Strategy	X	Context uses this interface to call the algorithm defined by a ConcreteStrategy	X
Concrete Strategy	The client class or other class except classes in the strategy pattern	Context uses this class which is passed through reference by client through polymorphism	Classes who hold the reference of ConcreteStrategy

Duck Game (Strategy)

Prof. Jonathan Lee (李允中)

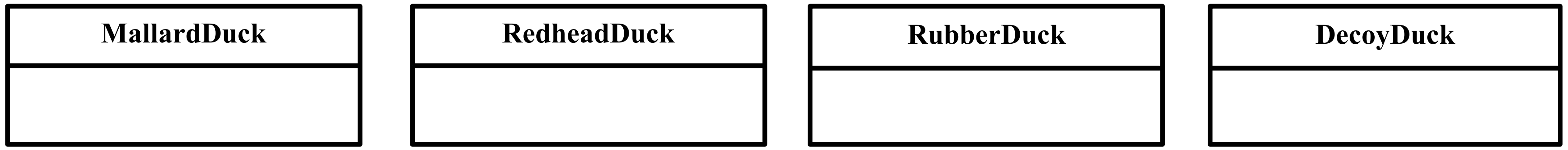
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statement

- There are four types of ducks in the game: MallardDuck, RedheadDuck, RubberDuck, and DecoyDuck.
- All types of the ducks have the same swim behavior but are with different displays.
- Some ducks can fly with wings, but some cannot fly.
- A duck can quack, squeak, or be silent.
- A duck can change its fly or quack behavior at run time.
- New fly or quack behaviors can be added, and the existing behaviors can be modified at compile time.

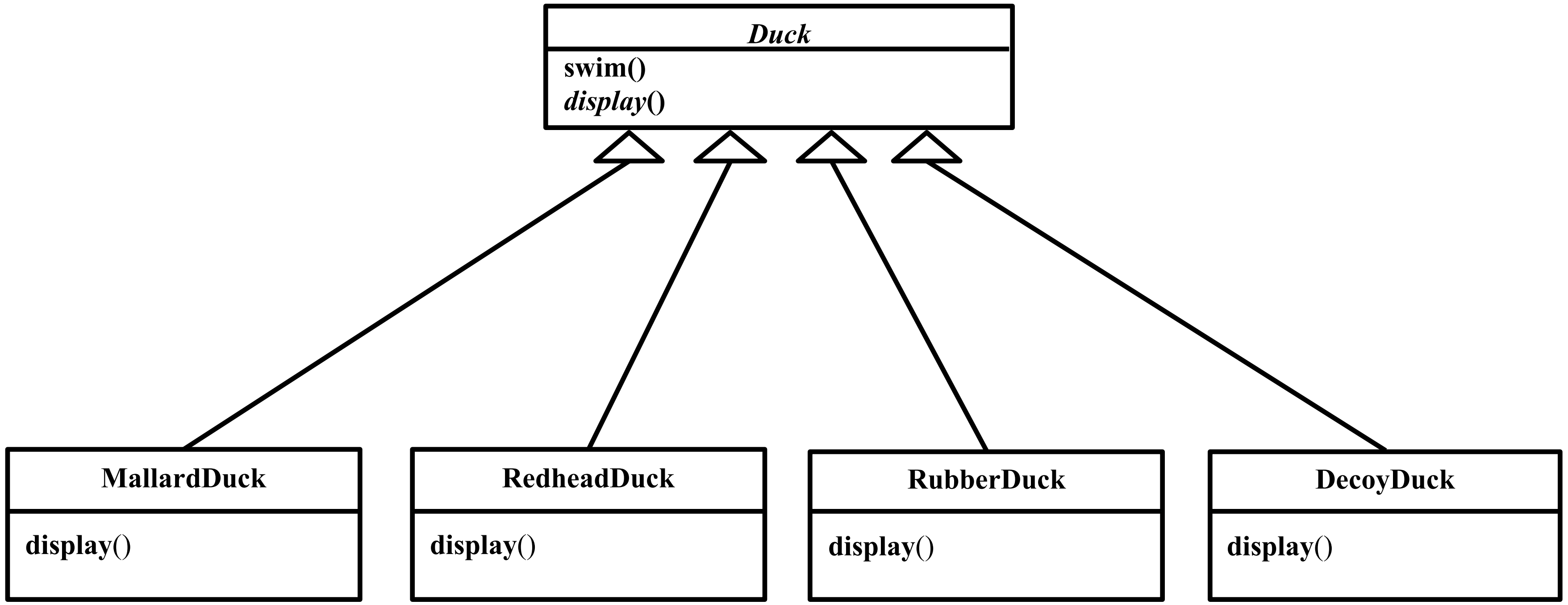
Requirements Statement$_1$

- There are four types of ducks in the game: MallardDuck, RedheadDuck, RubberDuck, and DecoyDuck.



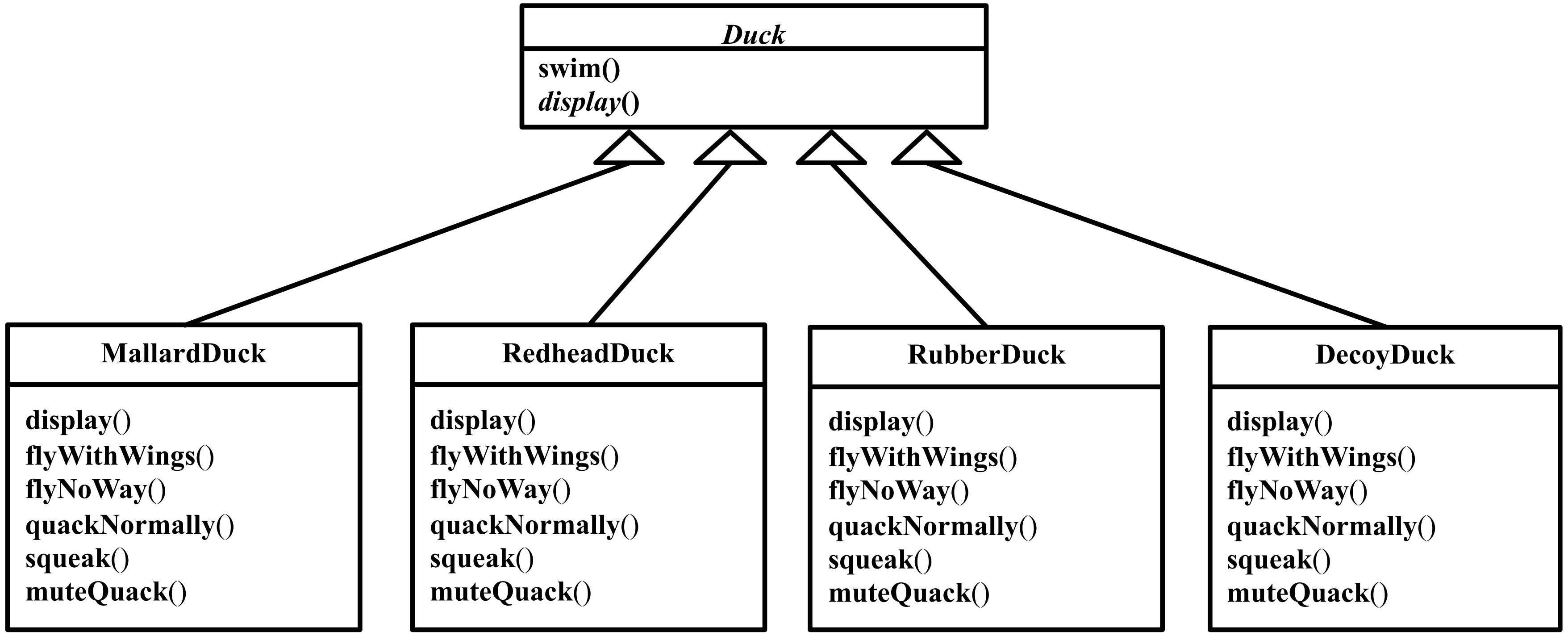
Requirements Statement$_2$

- All types of the ducks have the same swim behavior but are with different displays.



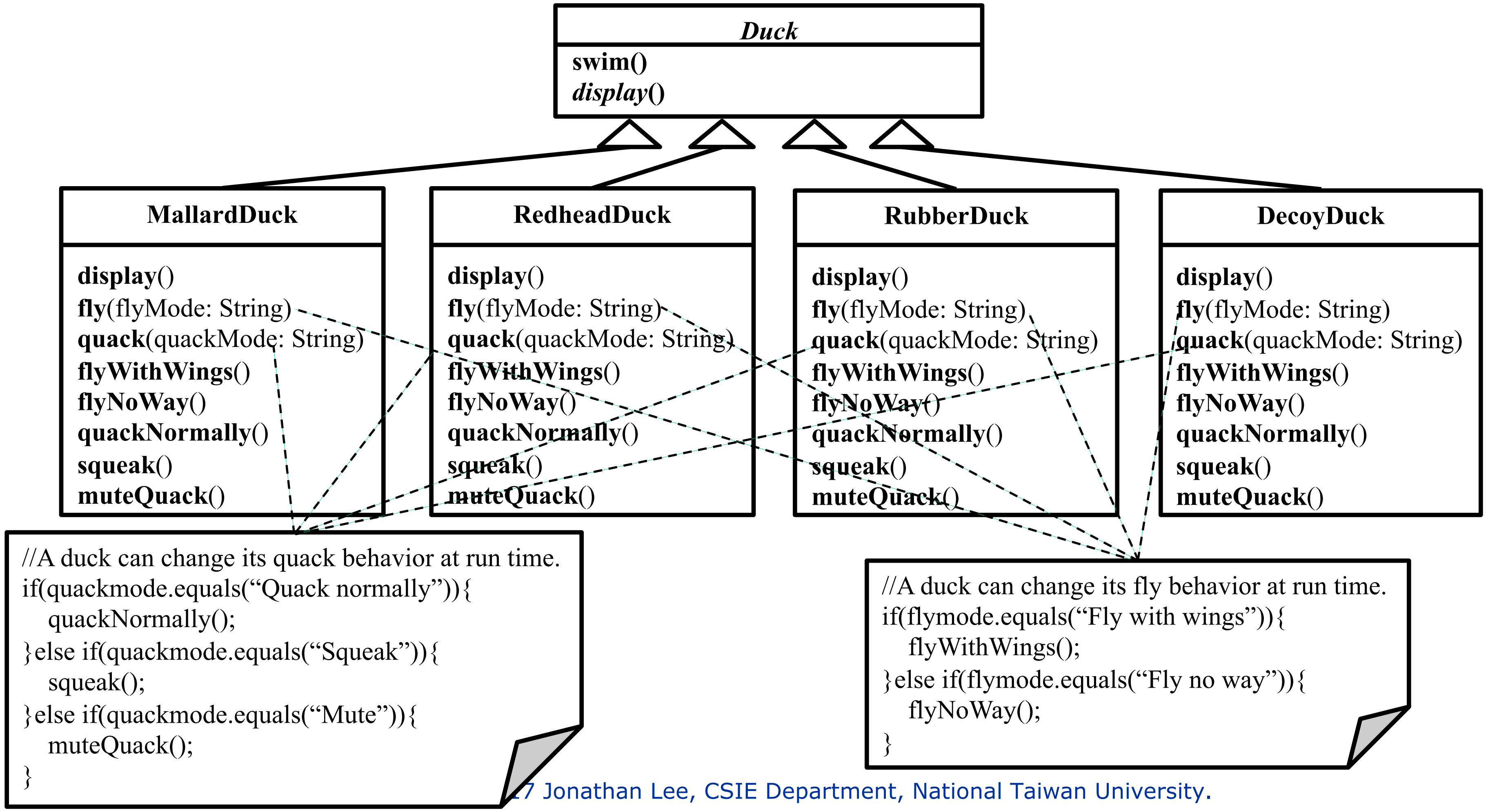
Requirements Statement$_3$

- Some ducks can fly with wings, but some cannot fly.
- A duck can quack, squeak, or be silent.



Requirements Statement$_4$

- A duck can change its fly or quack behavior at run time.



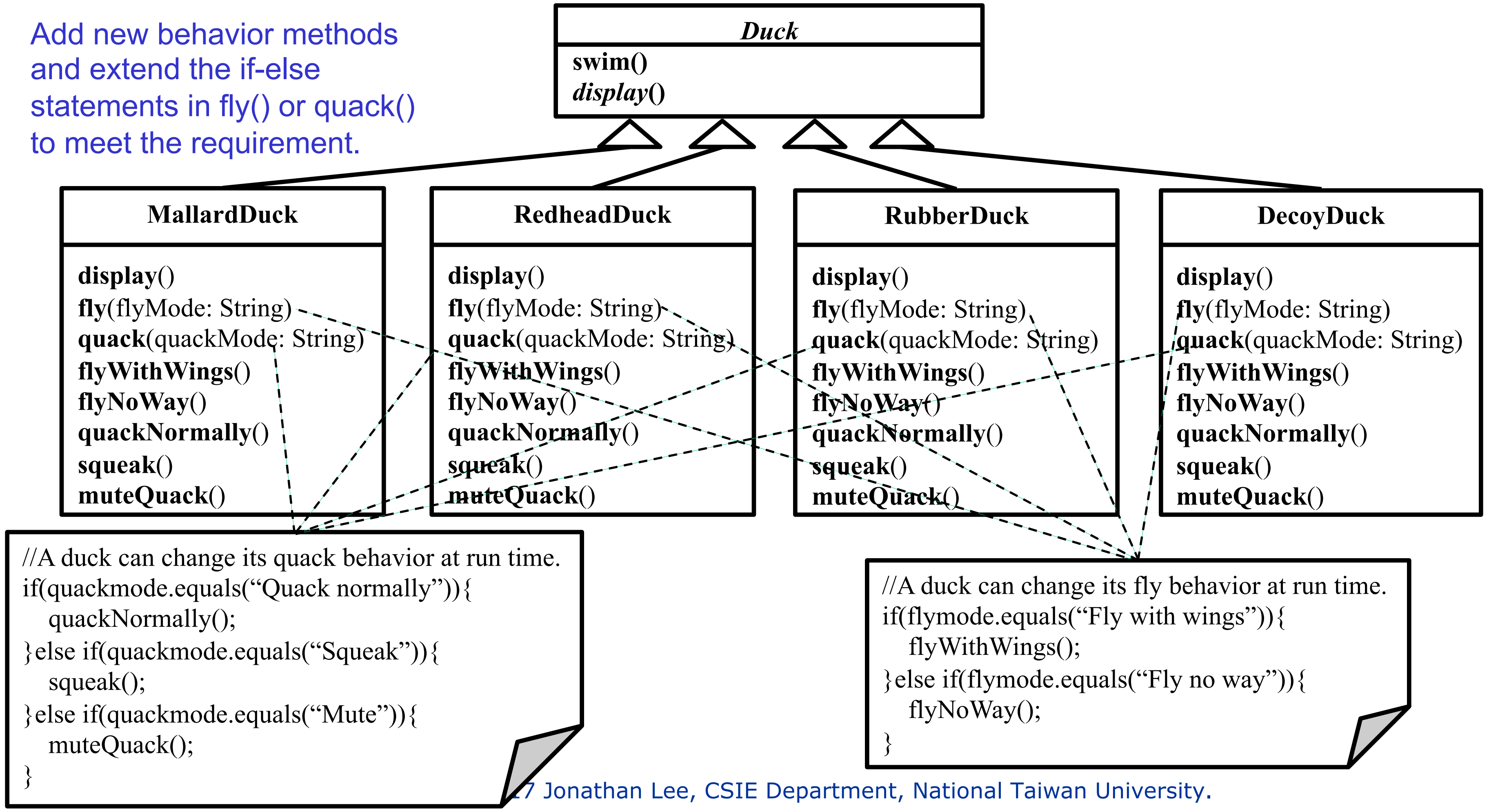
```
//A duck can change its quack behavior at run time.
if(quackmode.equals(“Quack normally”)){
   quackNormally();
}else if(quackmode.equals(“Squeak”)){
   squeak();
}else if(quackmode.equals(“Mute”)){
   muteQuack();
}
```

```
//A duck can change its fly behavior at run time.
if(flymode.equals(“Fly with wings”)){
   flyWithWings();
}else if(flymode.equals(“Fly no way”)){
   flyNoWay();
}
```

Requirements Statement$_5$

- New fly or quack behaviors can be added, and the existing behaviors can be modified at compile time.

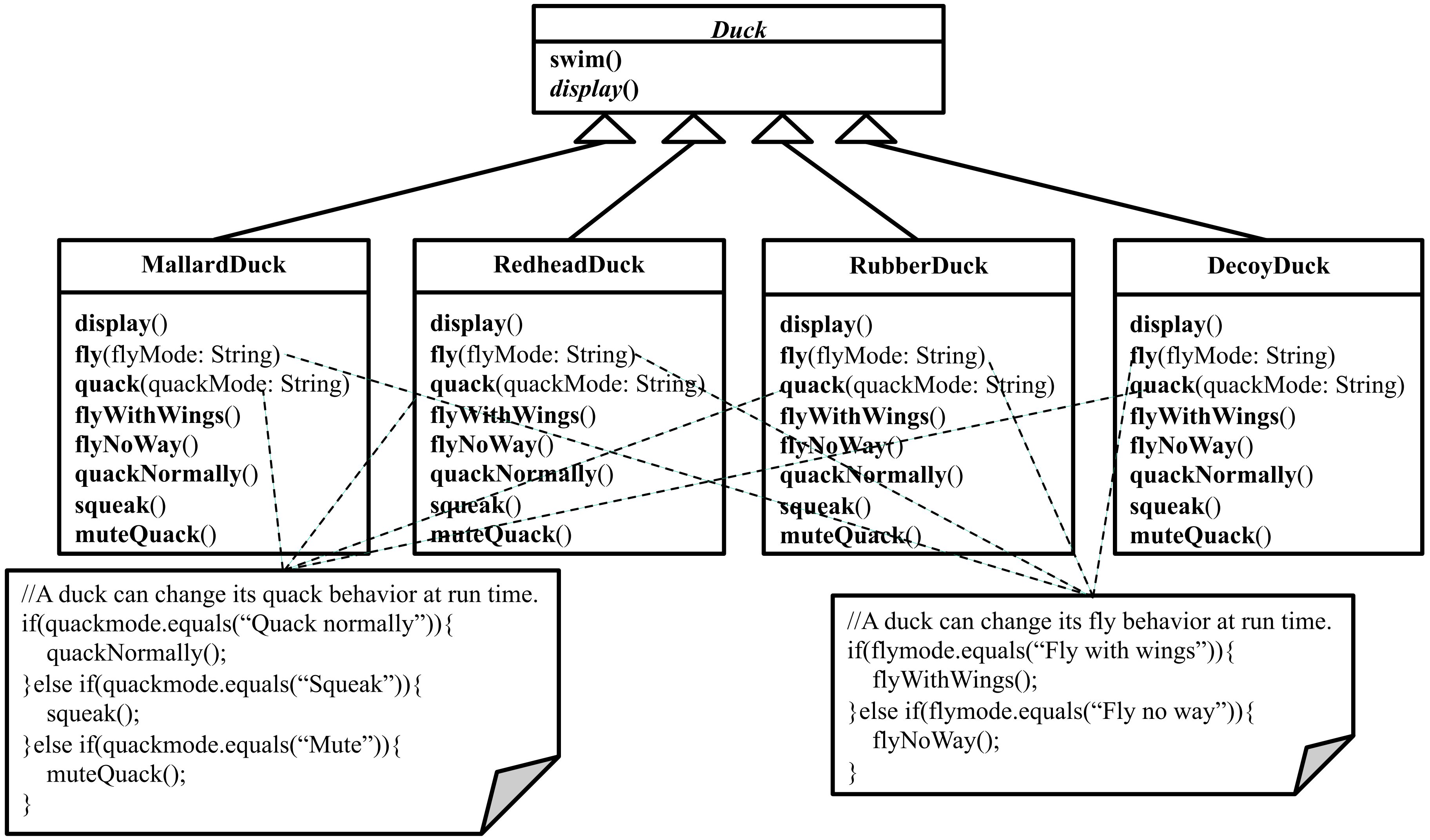
Add new behavior methods and extend the if-else statements in fly() or quack() to meet the requirement.



```
//A duck can change its quack behavior at run time.
if(quackmode.equals("Quack normally")){
   quackNormally();
}else if(quackmode.equals("Squeak")){
   squeak();
}else if(quackmode.equals("Mute")){
   muteQuack();
}
```

```
//A duck can change its fly behavior at run time.
if(flymode.equals("Fly with wings")){
   flyWithWings();
}else if(flymode.equals("Fly no way")){
   flyNoWay();
}
```

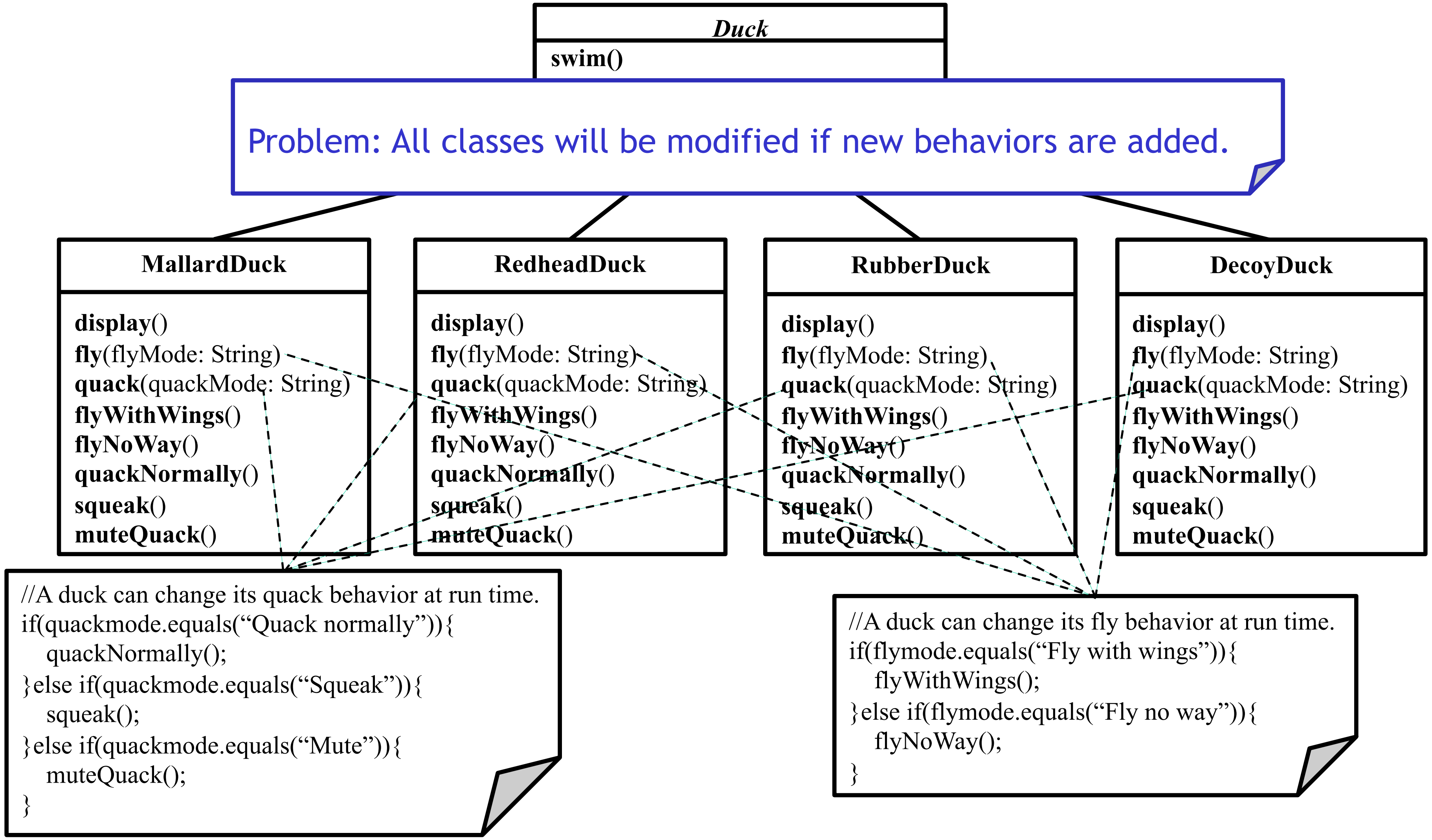
Initial Design - Class Diagram



```
//A duck can change its quack behavior at run time.
if(quackmode.equals("Quack normally")){
   quackNormally();
}else if(quackmode.equals("Squeak")){
   squeak();
}else if(quackmode.equals("Mute")){
   muteQuack();
}
```

```
//A duck can change its fly behavior at run time.
if(flymode.equals("Fly with wings")){
   flyWithWings();
}else if(flymode.equals("Fly no way")){
   flyNoWay();
}
```

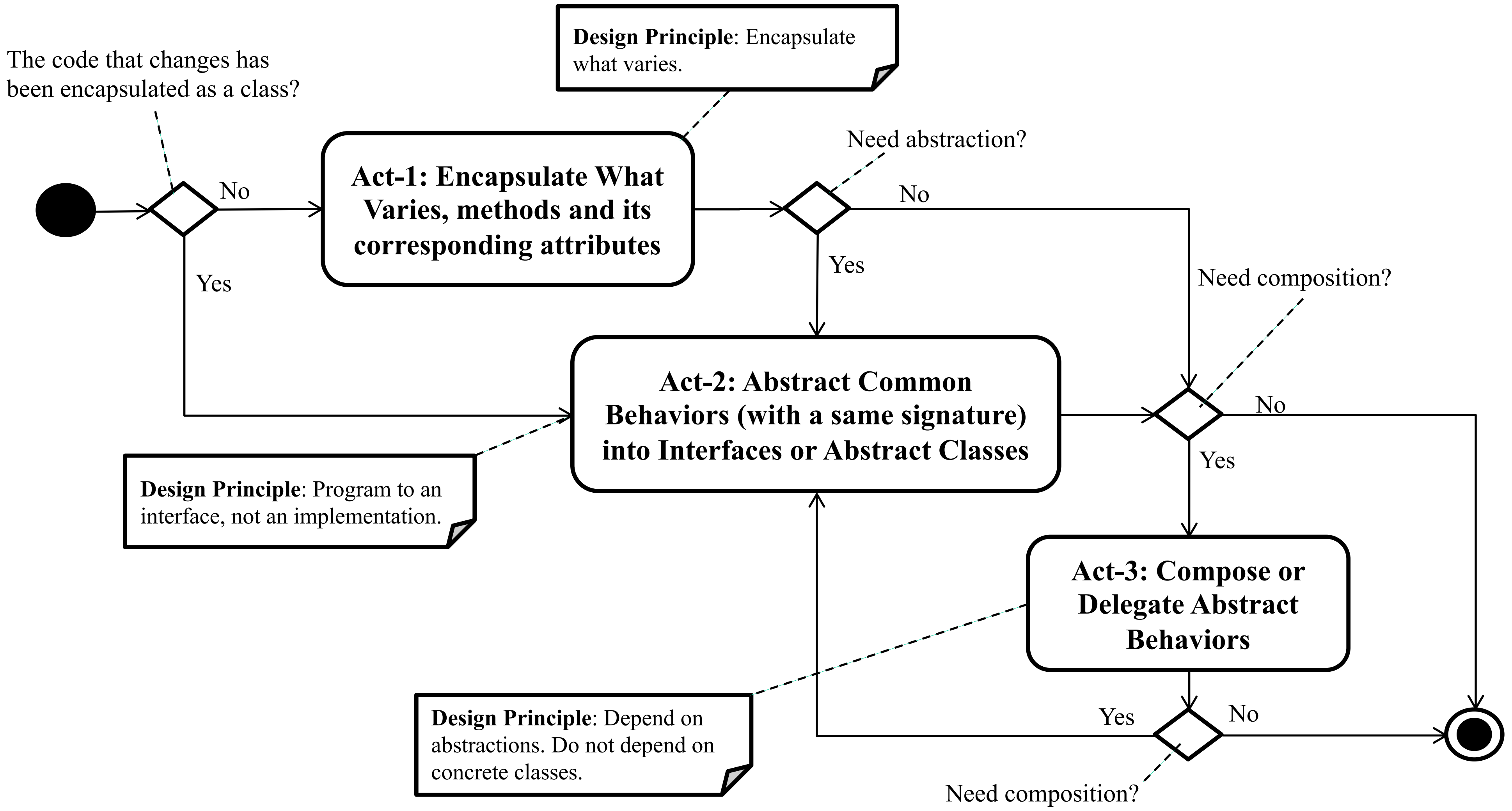
Problem with Initial Design



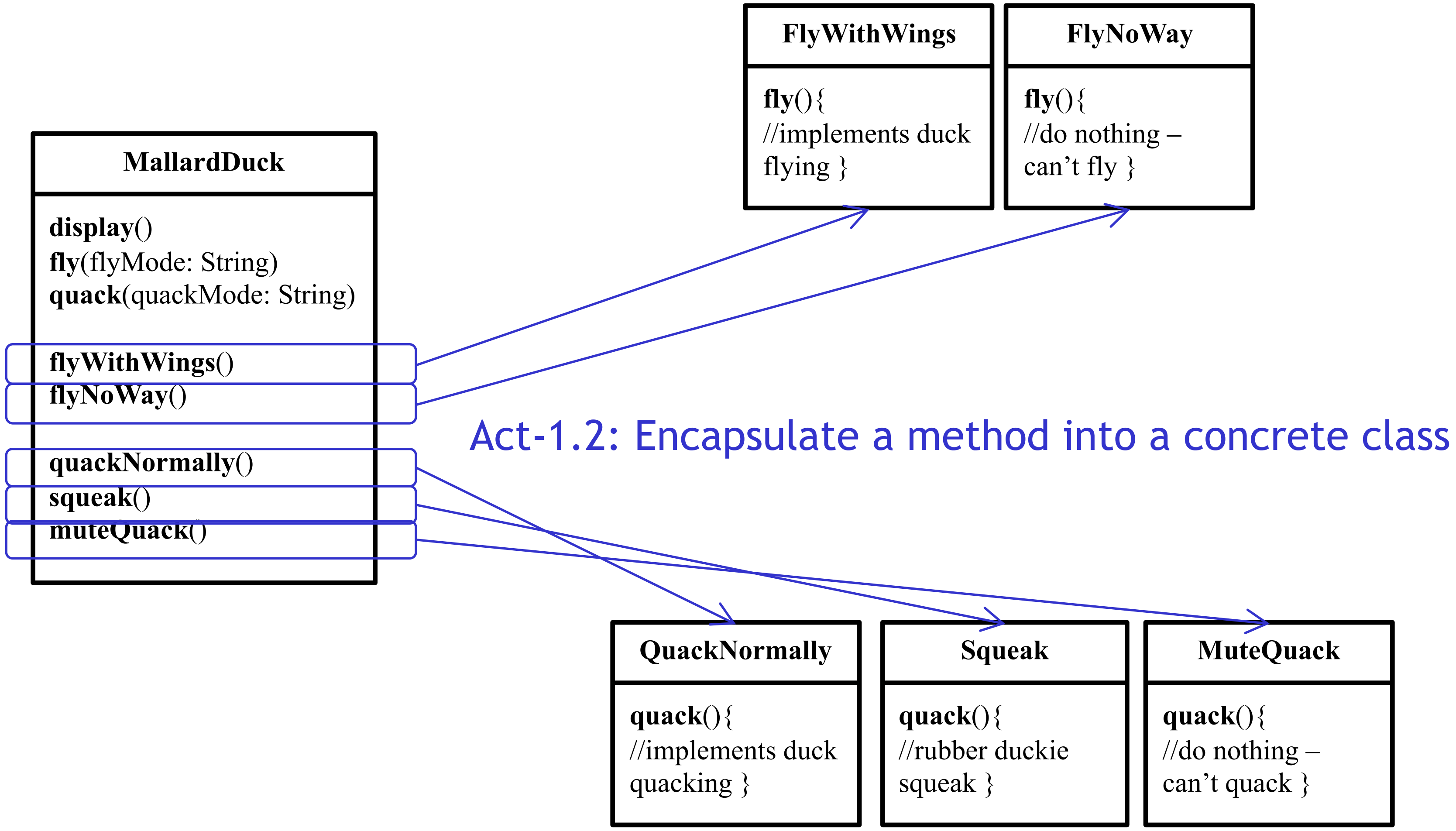
```
//A duck can change its quack behavior at run time.
if(quackmode.equals("Quack normally")){
   quackNormally();
}else if(quackmode.equals("Squeak")){
   squeak();
}else if(quackmode.equals("Mute")){
   muteQuack();
}
```

```
//A duck can change its fly behavior at run time.
if(flymode.equals("Fly with wings")){
   flyWithWings();
}else if(flymode.equals("Fly no way")){
   flyNoWay();
}
```

Design Process for Change

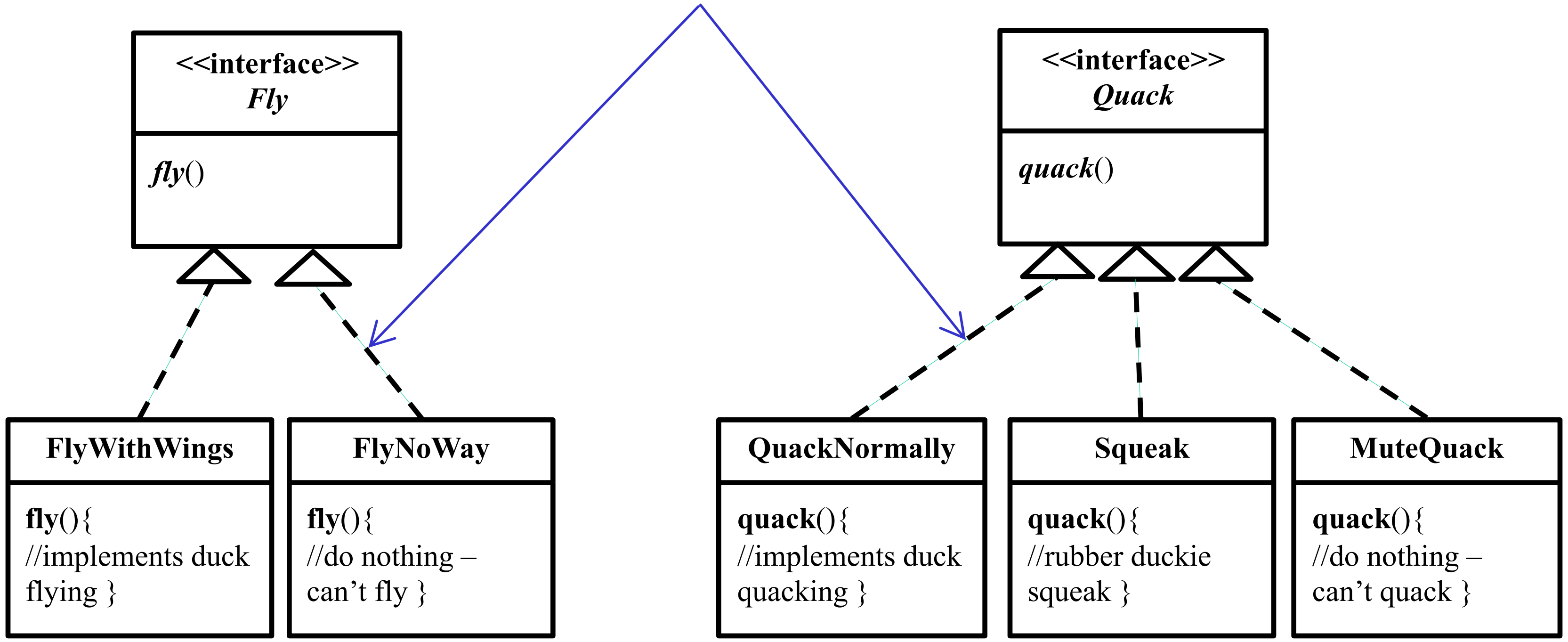

The code that changes has been encapsulated as a class?
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Design Principle: Encapsulate what varies.
Need abstraction?
No
Yes
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
Design Principle: Program to an interface, not an implementation.
Need composition?
No
Yes
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

Act-1: Encapsulate What Varies



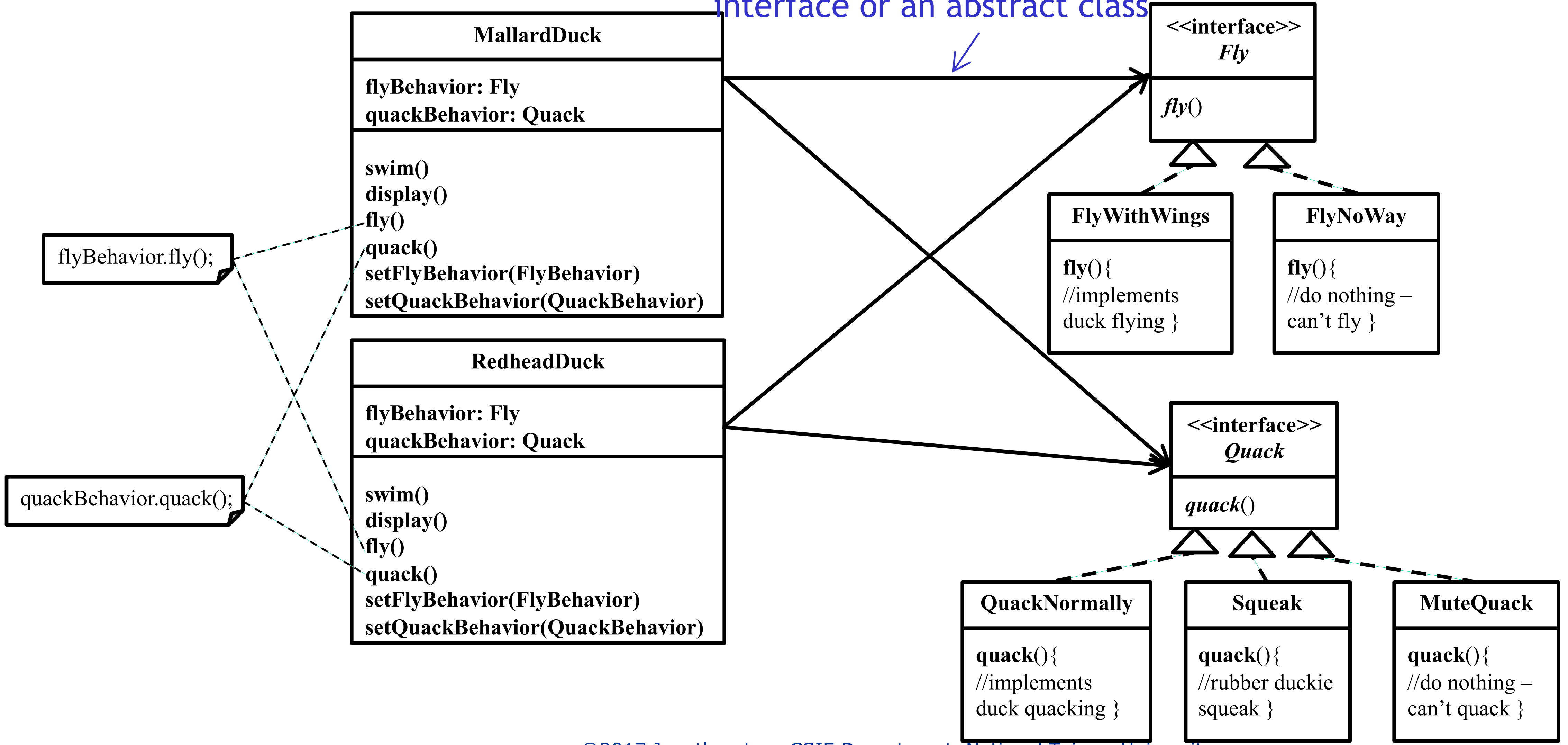
Act-2: Abstract Common Behaviors

Act-2.1: Abstract common behaviors with a same signature into interface through polymorphism



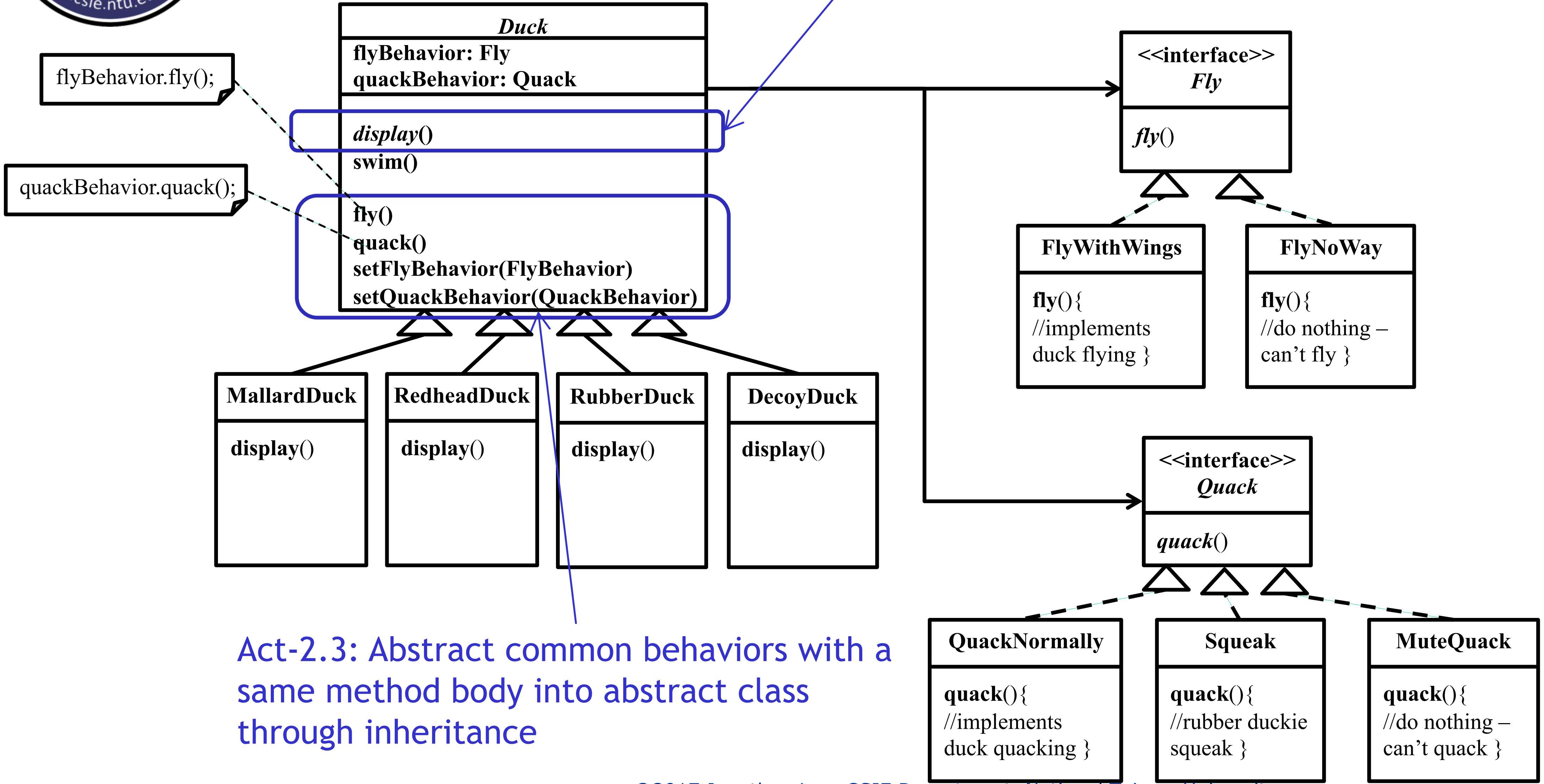
Act-3: Compose Abstract Behaviors

Act-3.1: Compose behaviors of an interface or an abstract class


MallardDuck
flyBehavior: Fly
quackBehavior: Quack
swim()
display()
fly()
quack()
setFlyBehavior(FlyBehavior)
setQuackBehavior(QuackBehavior)
RedheadDuck
flyBehavior: Fly
quackBehavior: Quack
swim()
display()
fly()
quack()
setFlyBehavior(FlyBehavior)
setQuackBehavior(QuackBehavior)
flyBehavior.fly();
quackBehavior.quack();
<<interface>> Fly
fly()
FlyWithWings
fly(){ //implements duck flying }
FlyNoWay
fly(){ //do nothing – can't fly }
<<interface>> Quack
quack()
QuackNormally
quack(){ //implements duck quacking }
Squeak
quack(){ //rubber duckie squeak }
MuteQuack
quack(){ //do nothing – can't quack }

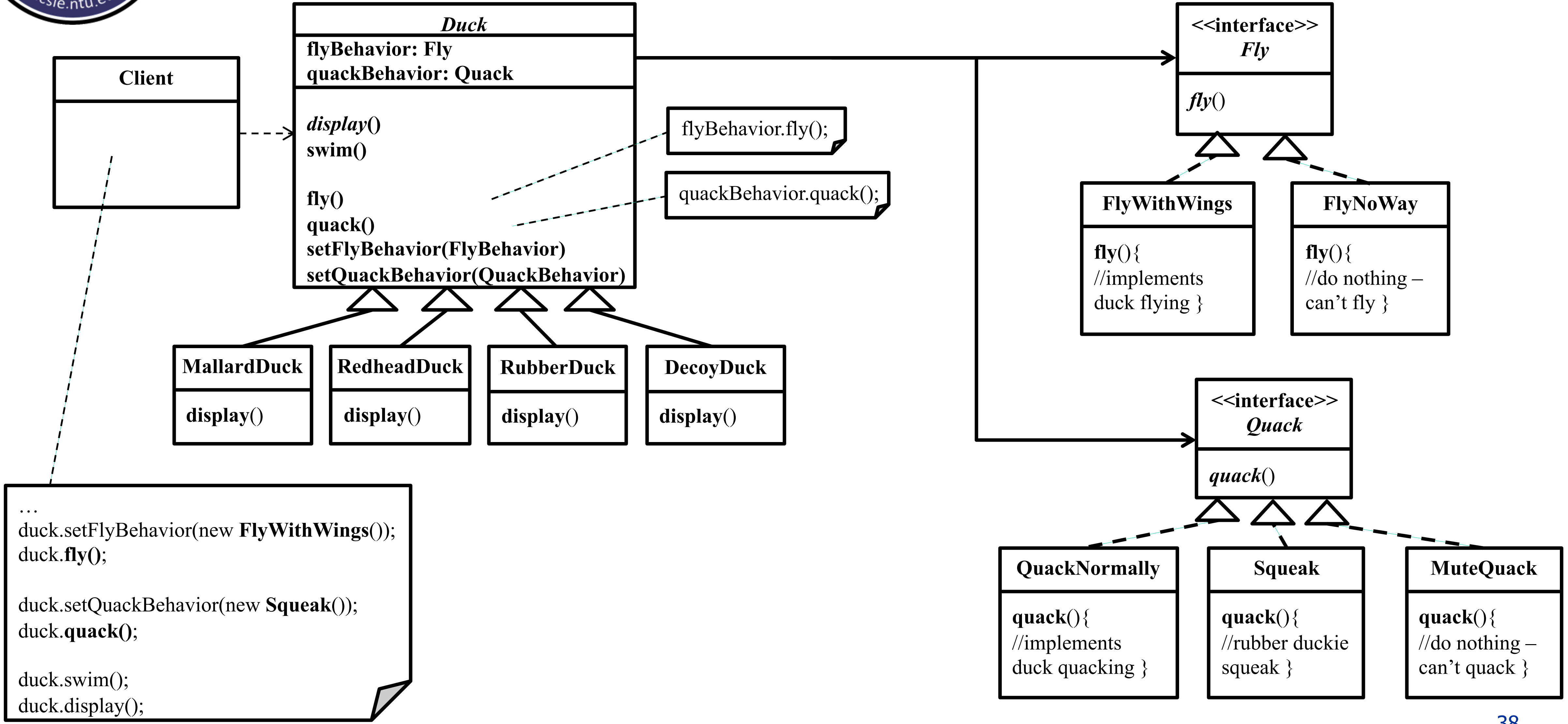
Act-2: Abstract Common Behaviors

Act-2.2: Abstract common behaviors with a same signature into abstract class through inheritance



Act-2.3: Abstract common behaviors with a same method body into abstract class through inheritance

Refactored Design after Design Process



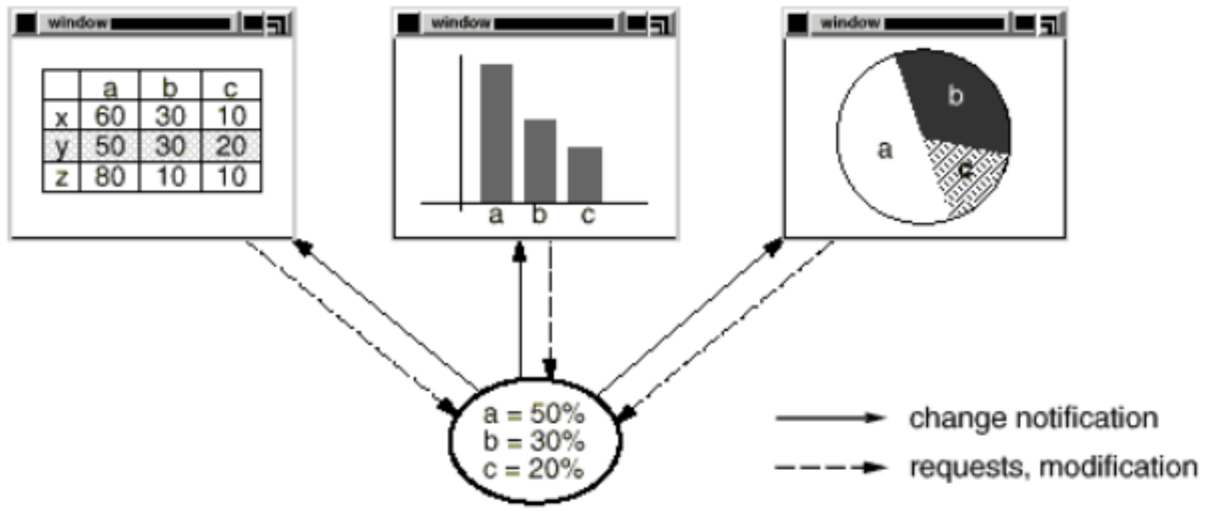
```
…
duck.setFlyBehavior(new FlyWithWings());
duck.fly();

duck.setQuackBehavior(new Squeak());
duck.quack();

duck.swim();
duck.display();
```

Homework 1: Requirements Statements$_1$

- In a spreadsheet application,
  - A spreadsheet object, bar chart object, and pie chart object can depict information in the same application data object by using different presentations.
  - When the user changes the information in the spreadsheet, the bar chart reflects the changes immediately, and vice versa.

Homework 1: Requirements Statements$_2$

- Both a spreadsheet object and bar chart object can depict information in the same application data object by using different presentations.


window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
a
b
c
a = 50%
b = 30%
c = 20%
change notification
requests, modification