SHARCNET™

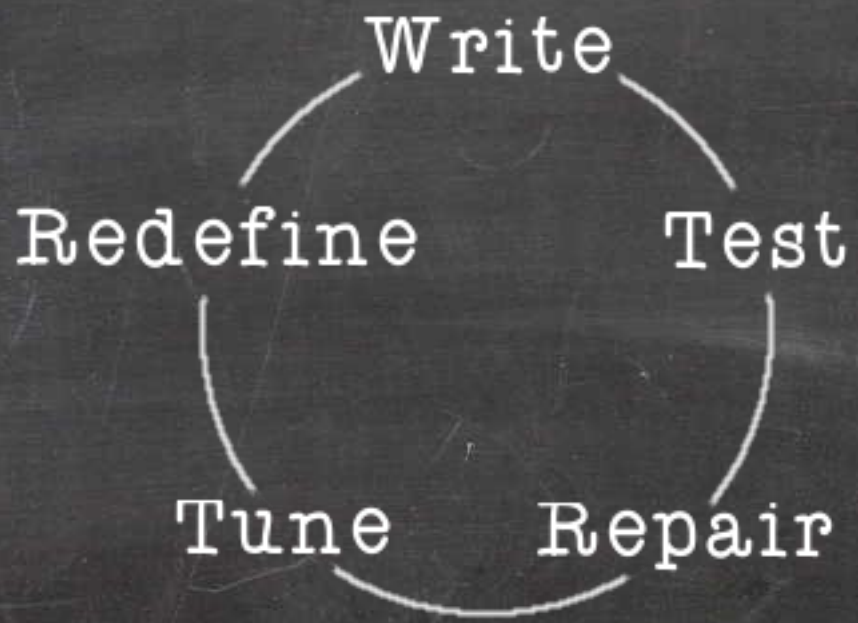
Defensive Programming Best Practices

Ed Armstrong

SHARCNET

December 2016

Development Process



Defensive Programming

The philosophy of writing code with the intention of handling unintended, and unforeseen errors, bugs, and circumstances.

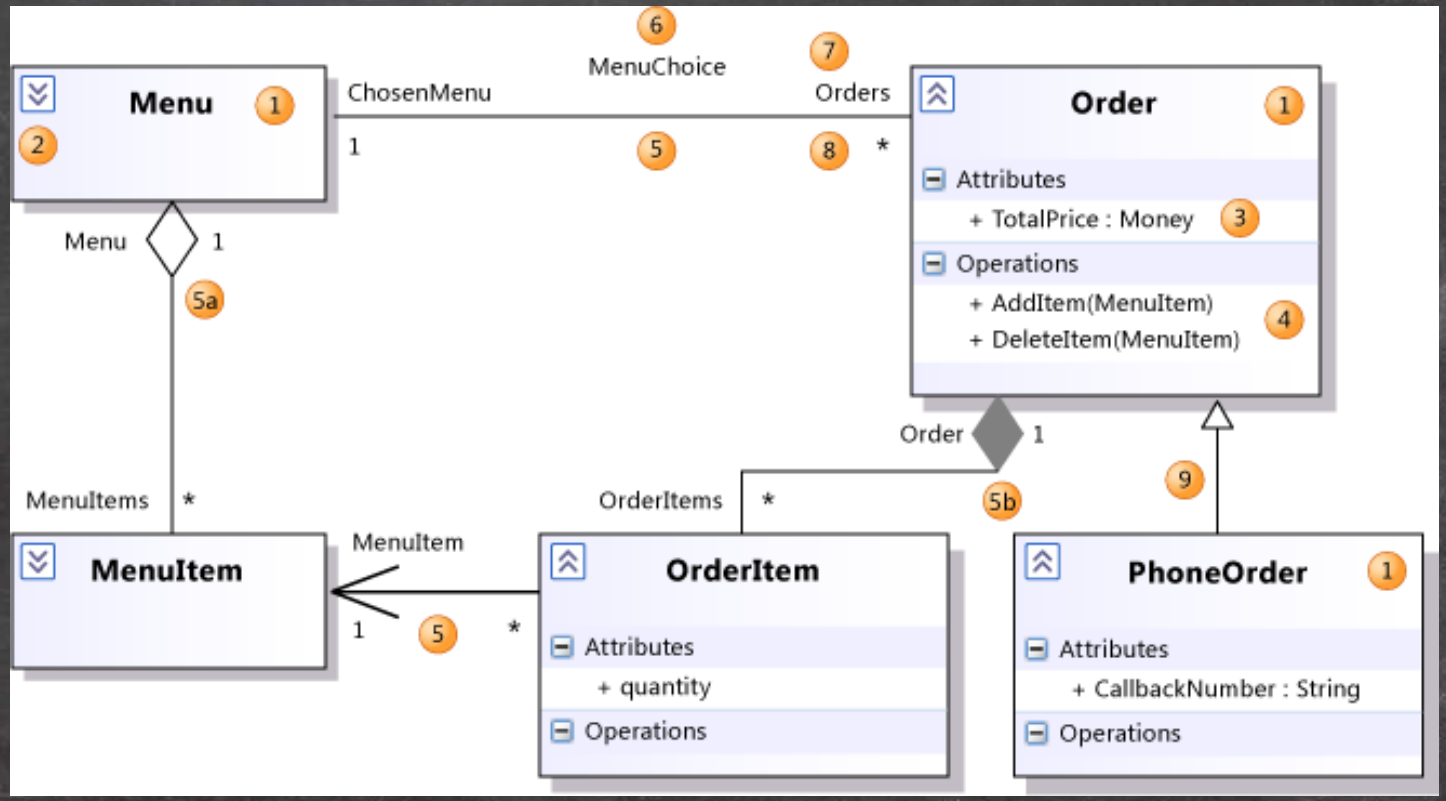
Goals of Defensive Programming

- **Correctness**
- **Clarity**
- **Readability**
- **Bug-Prevention**
- ~~**Speed**~~
- ~~**Early Optimization**~~
- ~~**Cleverness**~~

Part 1 : Object Oriented Programming

- UML
- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling
- Design Patterns

Unified Modeling Language (UML)

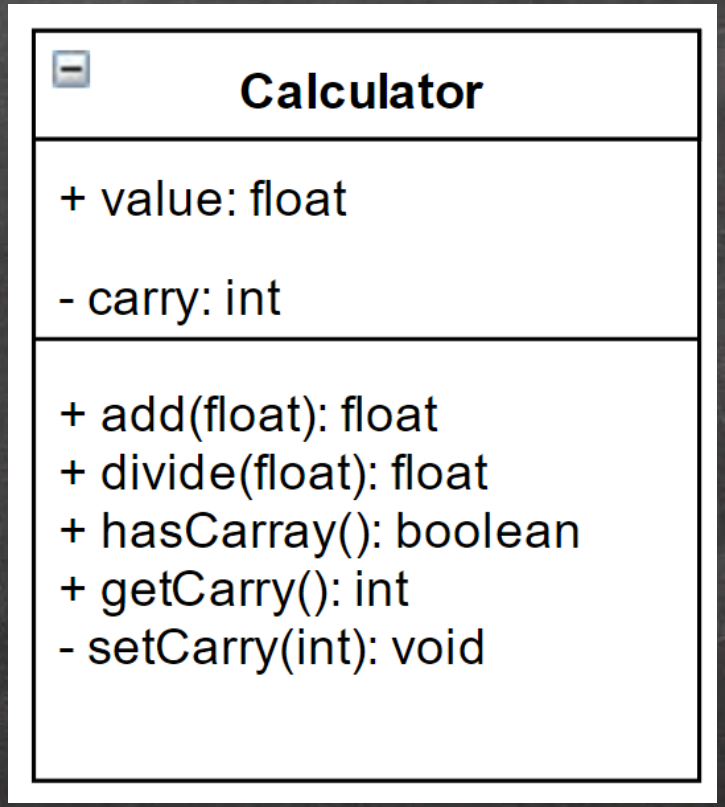


https://msdn.microsoft.com/en-us/library/dd409437.aspx

Object Oriented Programming

- **Abstraction**
- Encapsulation
- Inheritance
- Cohesion
- Coupling
- Design Patterns

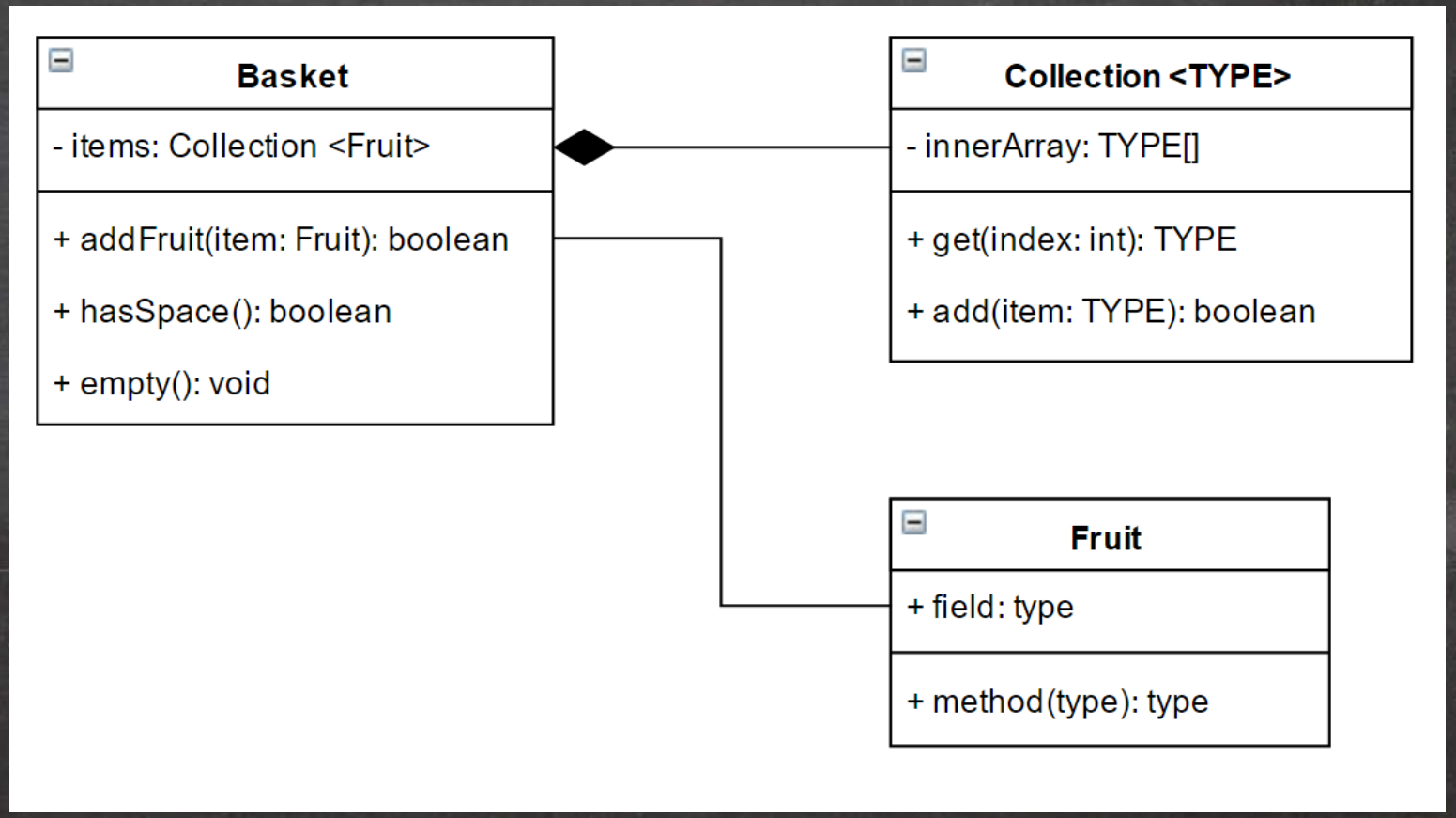
Reduce the complexity of an object by only exposing the relevant properties.



Object Oriented Programming

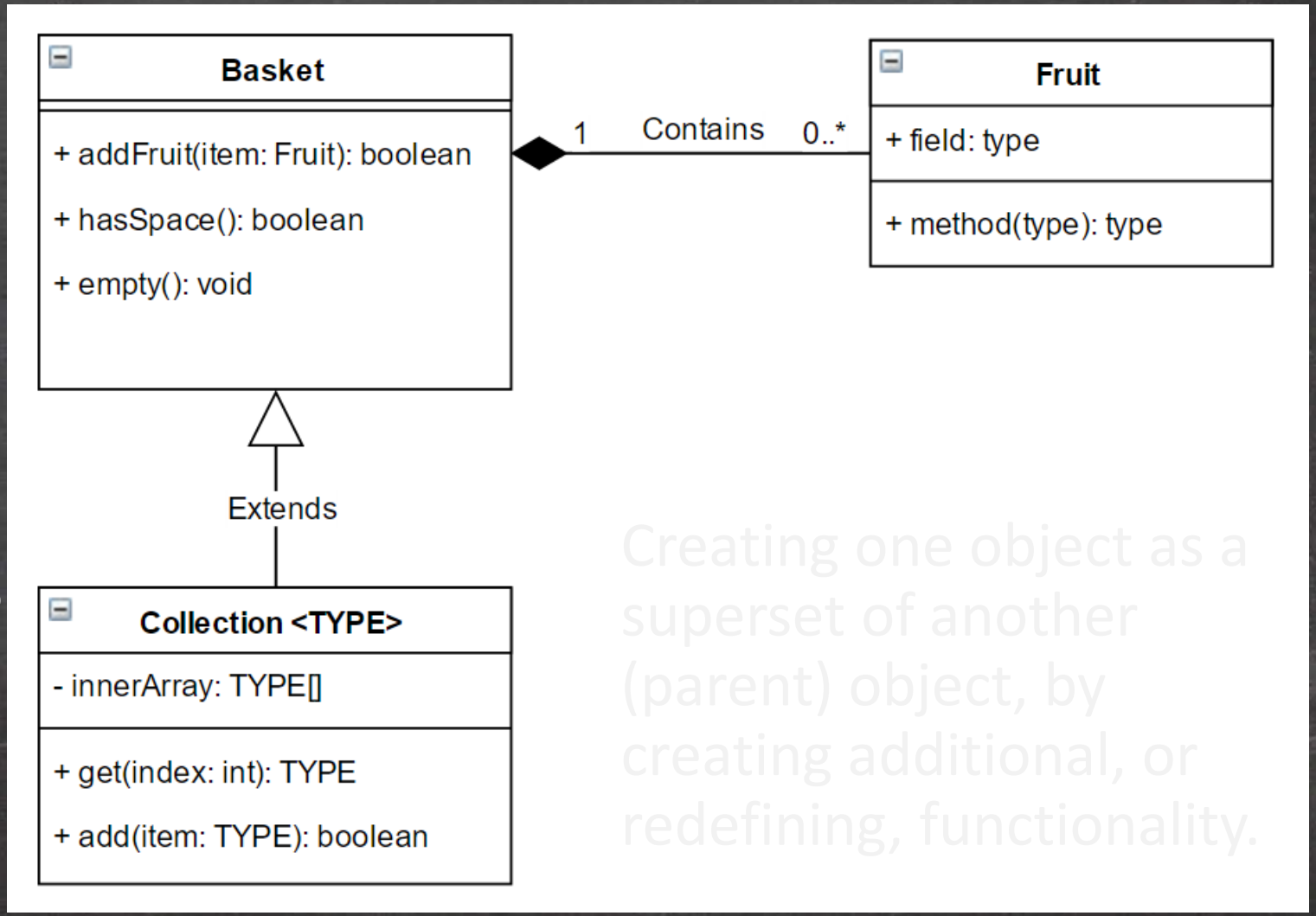
- Abstraction
- **Encapsulation**
- Inheritance
- Cohesion
- Coupling
- Design Patterns

Restricting access to an object's components through the use of methods.



Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling
- Design Patterns



Creating one object as a superset of another (parent) object, by creating additional, or redefining, functionality.

Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- **Cohesion**
- Coupling
- Design Patterns

The amount to which the components of an object relate to each other. The goal is to increase cohesion.

Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- **Coupling**
- Design Patterns

The amount to which two objects are dependent upon each other's behavior. The goal is to reduce coupling.

Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling
- **Design Patterns**

Formalized, and recognized practices for solving programming problems or designing an application.

Object Oriented Programming

- Abstraction
- Encapsulation
- Inheritance
- Cohesion
- Coupling
- **Design Patterns**

Structural Patterns	Behavioral Patterns
Composite	Null Object
Builder	Observer
Object Pool	Strategy
Singleton	Iterator

Parallel Patterns	Concurrency Patterns
Farmer-Worker	Thread Pool
Multi-Walk	Mutual Exclusion
Stream	Read-Write Lock
Map-Reduce	Monitor

Part 2 : Style

- modularity; avoid long code blocks
- use consistent naming conventions
- collections over arrays
- use iterators when available

Modularity : Avoid long code blocks

```
public void process (int i){
 /* long detailed comment on what
   * this it statement is doing */
 if (i == 0){
    blah…
    blah…
    blah…
  }
 /* long detailed comment on what
   * this it statement is doing */
  else if (i == 1){
    blah…
    blah…
    blah…
  }
 /* long detailed comment on what
   * this it statement is doing */
  else if (i == 2){
    blah…
    blah…
    blah…
  }
  … more else if statements ad nauseam
```

```
public void process (int i){
  if (i == 0) this.addAll();
  else if (i == 1) this.addSome();
  else if (i == 2) this.addNone();
  else this.addRandom();
}


public void addAll() …
public void addSome() …
public void addNone() …
public void addRandom() …
```

Naming Conventions

A set of rules for naming methods, variables, classes etc. which allows the reader to quickly identify the purpose of the particular identifier.

- letter separated words
- delaminated separated words
- pre/post-fix notation
  - symbol prefix (js __private)
  - word prefix (intSalary)
- rule bases
  - no numbers
  - noun / verb

camelCase
PascalCase
deliminated_words
ALLCAPS
ALL_CAPS_DELIM
intPrefix
$symbolPrefix

Naming Conventions : Readability

```
a = b * c;
pay = wage * hours;
```

```
for (int i = 0; i < n; i++){
    object = array[i];
}
```

```
interface ICollection{…}
abstract ACollection{…}
interface HasContext{…}
```

```
for (int x = 0; x < width; x++)
    for (int y = 0; y < height; y++)
```

```
class{
    setWage(float perHourWage){
        this.wage = wage;
    }

    setHours(int hoursWorked){
        this.hours = hoursWorked;
    }

    float getPay(){
        return wage * hours;
    }
}
```

```
package fruit;

import java.awt.Shape;

public class Fruit implements IFood{
   static final int DISPLAY_RESOLUTION = 640;
   private Flavour flavour;
   private Colour colour;
   private Shape shape;

   public Fruit(){ ... }
   public void ripen(){ ... }
   public void addToBasket(Basket basket){ ... }
}
```

Package

Package Tree

Class/Interface

"Global"

Variable

Method

Multi-word Names

```
// Can you spot the bug?

enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext();
                deck.add(new Card(i.next(), j.next()));
```

Code snippet from “Essential Java 2nd Edition”

```
// Can you spot the bug?

enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

List<Card> deck = new ArrayList<Card>();

for (Iterator<Suit> i = suits.iterator(); i.hasNext(); )
        for (Iterator<Rank> j = ranks.iterator(); j.hasNext();
                deck.add(new Card(i.next(), j.next()));
```

Code snippet from “Essential Java 2nd Edition”

```
enum Suit { CLUB, DIAMOND, HEART, SPADE }
enum Rank { ACE, DEUCE, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT,
NINE, TEN, JACK, QUEEN, KING }
...
Collection<Suit> suits = Arrays.asList(Suit.values());
Collection<Rank> ranks = Arrays.asList(Rank.values());

for (Suit suit : suits){
  for (Rank rank : ranks){
    deck.add(new Card(suit, rank));
  }
}
```

Code snippet from “Essential Java 2nd Edition”

Part 3: Patterns and Practices

- exceptional circumstances
- null values
- equals
- interfaces
- strategy pattern
- stateless objects

Exceptional Circumstances

- **Assert** : Things that should never occur; to protect against programming errors (private methods, return value validation).
- **Exception** : Uncommon conditions out of the programmer's control, that the end user may want to know about or requires special handling. (Missing files, dropped connections, etc).
- **Return Value:** Use when a situation isn’t terminal.

NULL is a four letter word!

I call it my billion-dollar mistake ...the invention of the null reference in 1965. At that time, I was designing [ALGOL] ... I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

- Richard Hoare, the inventor of the NULL reference.

Guidelines to using NULL

- Only use null privately in a class.
- Don't return NULLs.
- Throw an exception when null is passed in.
- Return empty collections (or zero length arrays).
- Return NULL representative objects.

Return Empty Collections

in class
```
public Fruit[] getFruit(){
  if (collection.size() == 0)
  return null;
  …
}
```

in main
```
Fruit f = getFruit();
if (f != null){
   for (Fruit f : getFruit()) {
      [do something]
   }
}
```

in class
```
public Fruit[] getFruit(){
  if (collection.size() == 0)
  return new Fruit[0];
  …
}
```

in main
```
for (Fruit f : getFruit()) {
   [do something]
}
```

Single Instance

```
class Basket{
   private ArrayList <Fruit> fruits;
   public Fruit getFruit(){
      if (fruits.size() == 0)
         return null;
      else
         return fruits.remove(0);
   }
}
```

Single Instance Fixed

```
class Basket{
   private ArrayList <Fruit> fruits;

   public boolean hasFruit(){
      return fruits.size() > 0;
   }

   public Fruit getFruit(){
      if (fruits.size() == 0) throw new NullPointerException();
      else return fruits.remove(0);
   }
}
```

NULL misrepresenting errors

```
Fruit aPieceOfFruit = fruitBasket.getFruit();
[do some stuff]
aPieceOfFruit.eat(); /* throws exception */
```

NULL misrepresenting errors

```
Fruit fruit = basket.getFruit(); /* throws exception */
[do some stuff]
aPieceOfFruit.eat();
```

Interfaces

```
interface IFood {
    Flavour getFlavour();
    Size getSize();
    void setPortions(int i);
    int getPortions();

    default void bite(){
        this.setPortions(this.getPortions() - 1);
    }
}
```

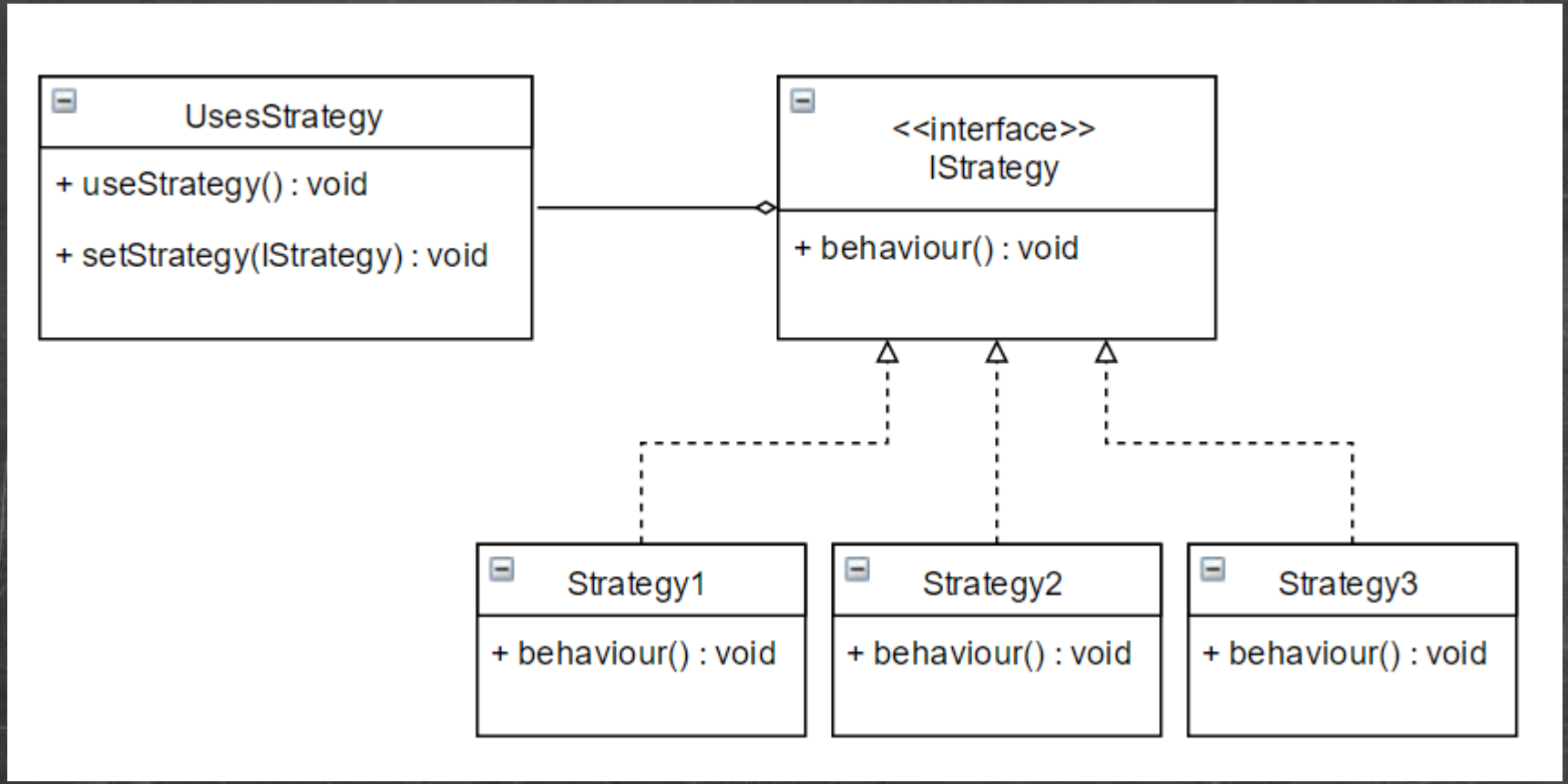
Code to the Interface

```
Class Bucket{
        private ArrayList myList= new ArrayList();
}

Class Bucket{
        private List myList = new ArrayList();
}

Class Bucket{
        private List myList = new TreeList();
}
```

Strategy Pattern



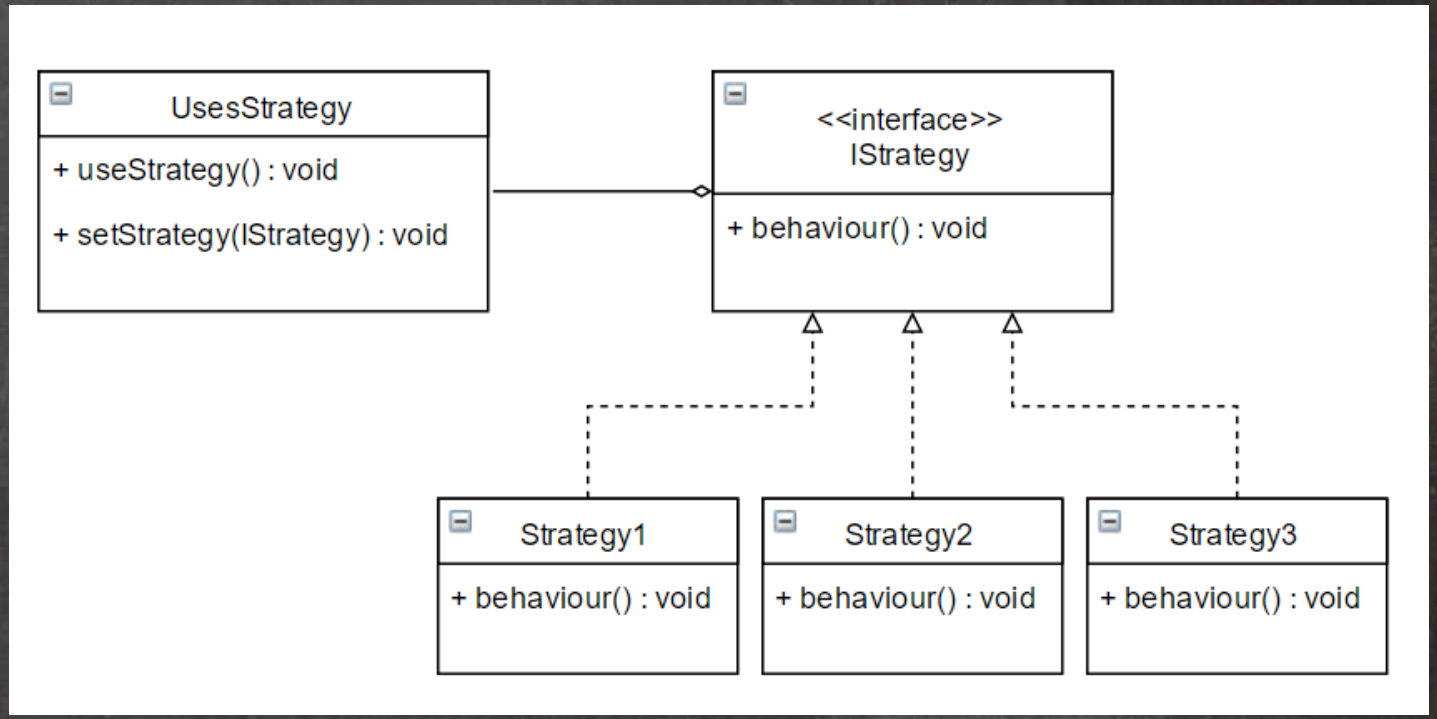
Strategy Pattern

```
interface IStrategy{
	public void behaviour();
}

class Strategy1 implements IStrategy{
	public void behaviour(){…}
}

class Strategy2 implements IStrategy{
	public void behaviour(){…}
}

class Strategy3 implements IStrategy{
	public void behaviour(){…}
}
```

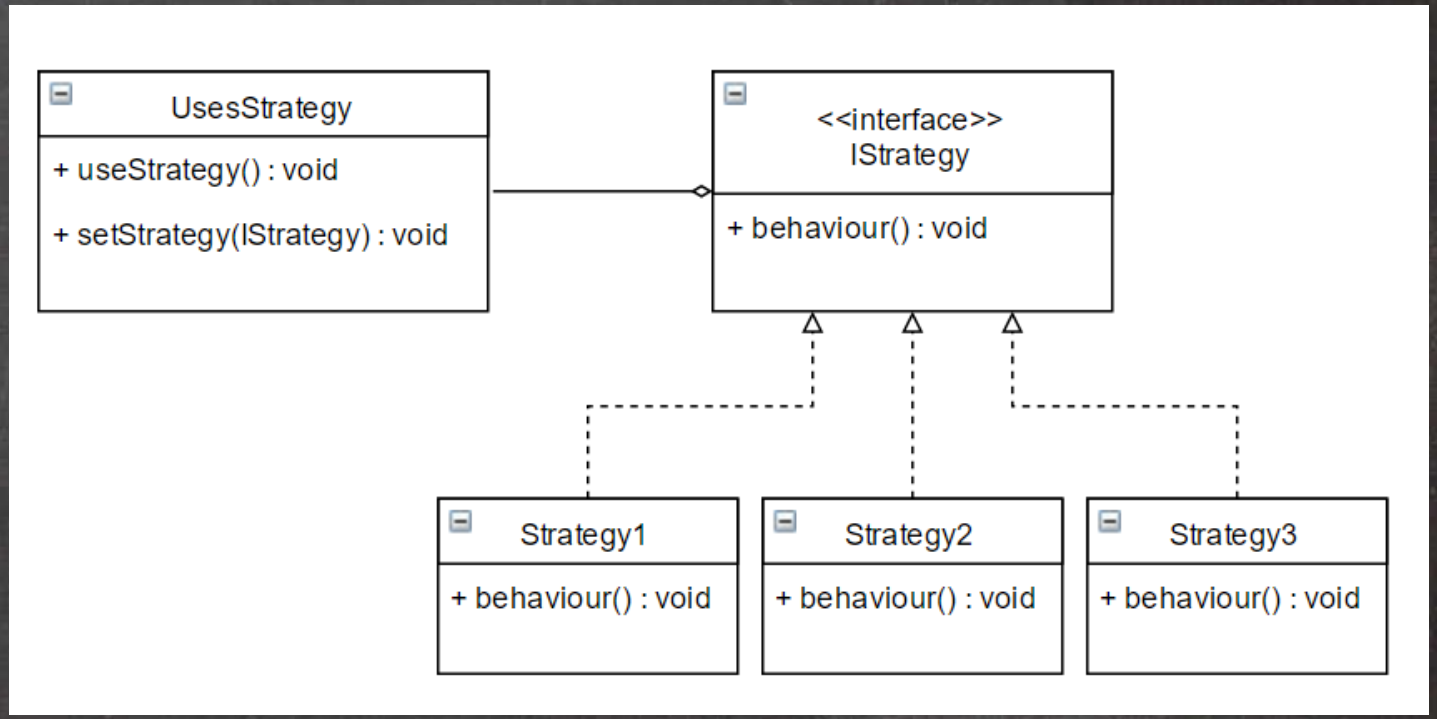


Strategy Pattern

```
class UsesStrategy {
    private IStrategy strategy;

    public void setStrategy(IStrategy strategy){
        this.strategy = strategy;
    }

    public void useStrategy(){
        … stuff …
        useStrategy();
        … stuff …
    }
}
```



Immutable Pattern

```
public interface ImmutableFruit {
   Flavour getFlavour();
   Colour getColour();
   Shape getShape();
}
```

```
public class Fruit
implements ImmutableFruit{

   private Flavour flavour;
   private Colour colour;
   private final Shape shape;

   public Flavour getFlavour() {...}
   public Colour getColour() {...}
   public Shape getShape() {...}

   public Fruit(){ ... }

   public void ripen(){ ... }
}
```

Citations

- Collected Java Practices,. [online], http://www.javapractices.com (accessed 2015)

- Bloch, J. "Effective Java, 2nd edn. The Java Series." (2008).