Software Design Patterns: Session Three

Sahar Mostafa
UC Berkeley Extension
Summer 2020

Ice Breaker - Hobbies

Tell us about a hobby of yours or a passion you have

Week In Review

1. Reusability
2. Refactoring
3. Design Patterns Categories
4. Code Reviews and Exercises

Software Design Patterns

Creational

Singleton

Only one instance is allowed in the system. For example: DB connection or a logger utility

Factory

Create objects based on a type. Actual instatiation is delegated to subclasses

Builder

The builder hides the process of building objects with complex structure, separates the representation of the object and its construction

Prototype

Useful when dealing with cloning objects and dynamic loading of classes

Structural

Adapter

Allow two incompatible interfactes to communicate by converting interfaces

Decorater

Used when there is a need to enhance or extend the behavior of an object dynamically

Facade

A single uber interface to one or more subsystems or interfaces

Proxy

Represents another object and can act on its behalf shielding an object from direct interaction

Behavioral

State

Mutating data allowing an object to alter behavior when its internal state changes

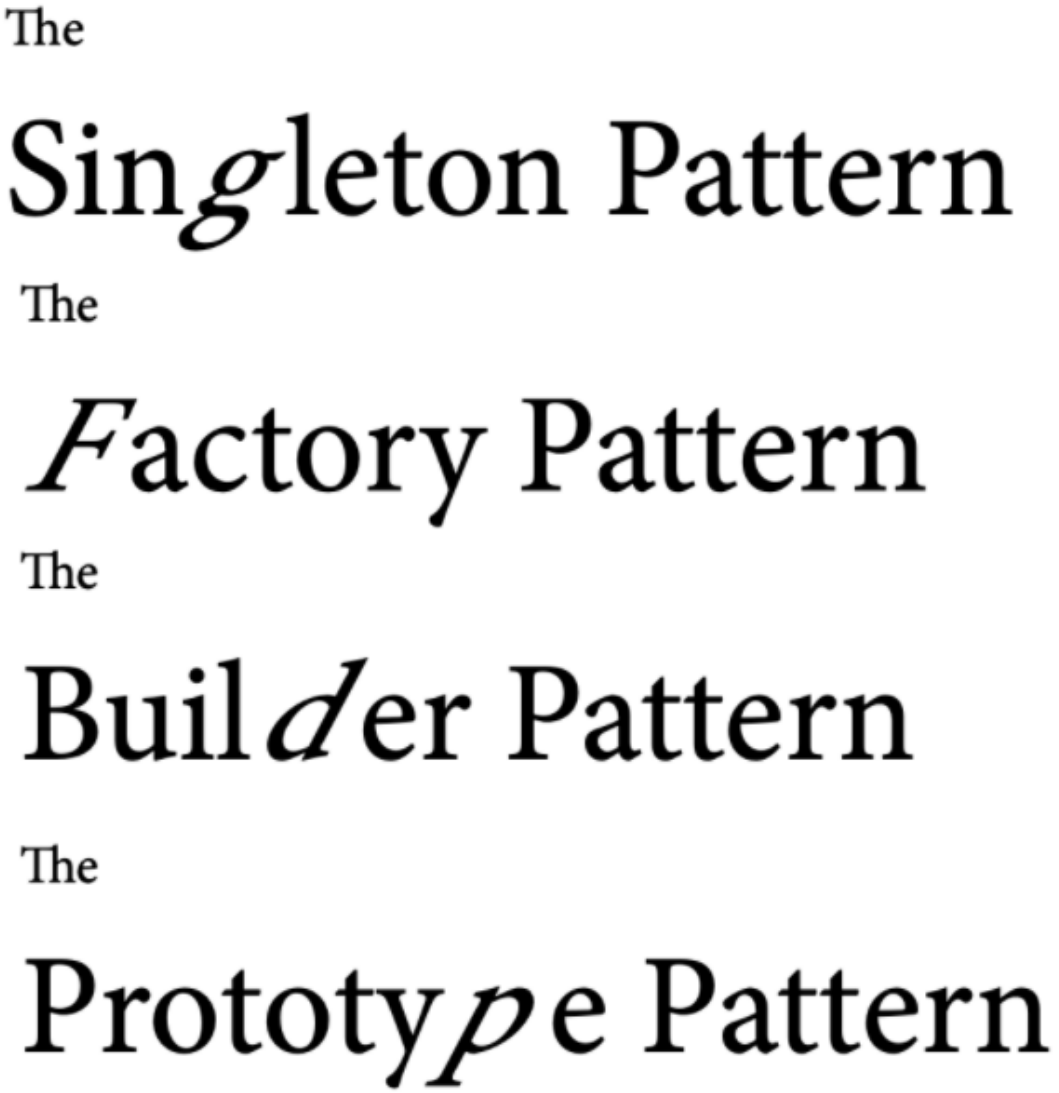
Observer

USed when one object is interested in the state of other dependent objects. When one object changes state all the dependents are notified

Strategy

A common interface with differnet internal algorithms allowing switching out one algorithm for another seamlessly.

Creational Patterns



Description

Intent - Abstract Instantiation

System is independent of how objects are created

Shift emphasis from hard coded set of behaviors toward a set of fundamental behaviors

Control over *what* gets created *who* creates and *how* it gets created

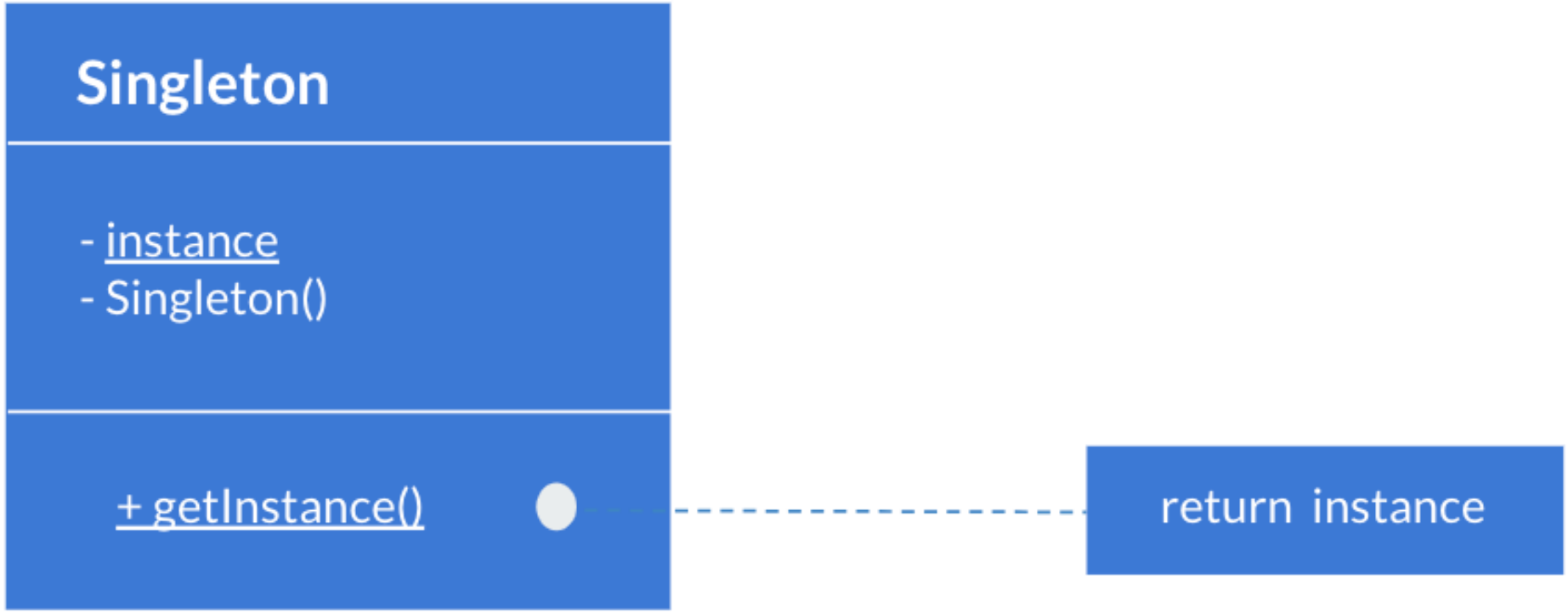
Construction

Singleton Pattern

Description

Intent	Motivation	Consequences	Use cases
Separate construction from representation	Same construction can be used for different representations Creating a complex object independently	Variable internal representations Finer control over construction Encapsulates code for construction and representation	Reader for different document types, building an aeroplane with different parts

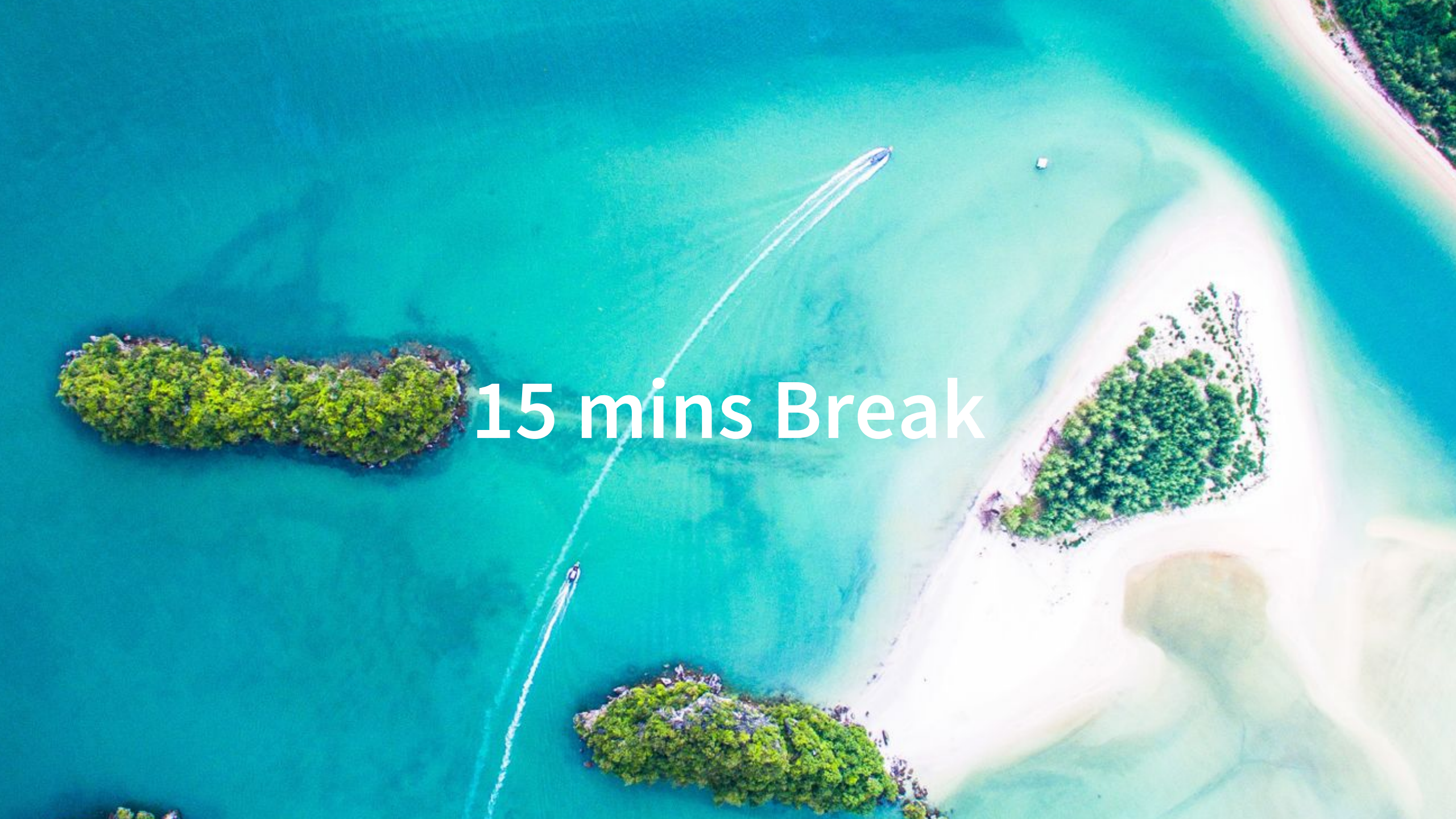
UML Diagram


Singleton
- instance
- Singleton()
+ getInstance()
return instance

Implementation

```
class Singleton {
    public:
       static Singleton* Instance();
    private:
       Singleton();
       static Singleton* _instance;
};

Singleton* Singleton::_instance = NULL;

Singleton* Singleton::getInstance() {
    if (_instance = NULL) {
       _instance = new Singleton();
    }
    return _instance;
 }
```

```
public class Singleton {
 private static final Singleton instance = new
Singleton();

 private Singleton(){}

 public static Singleton getInstance()
  return instance;
 }
}
```


15 mins Break

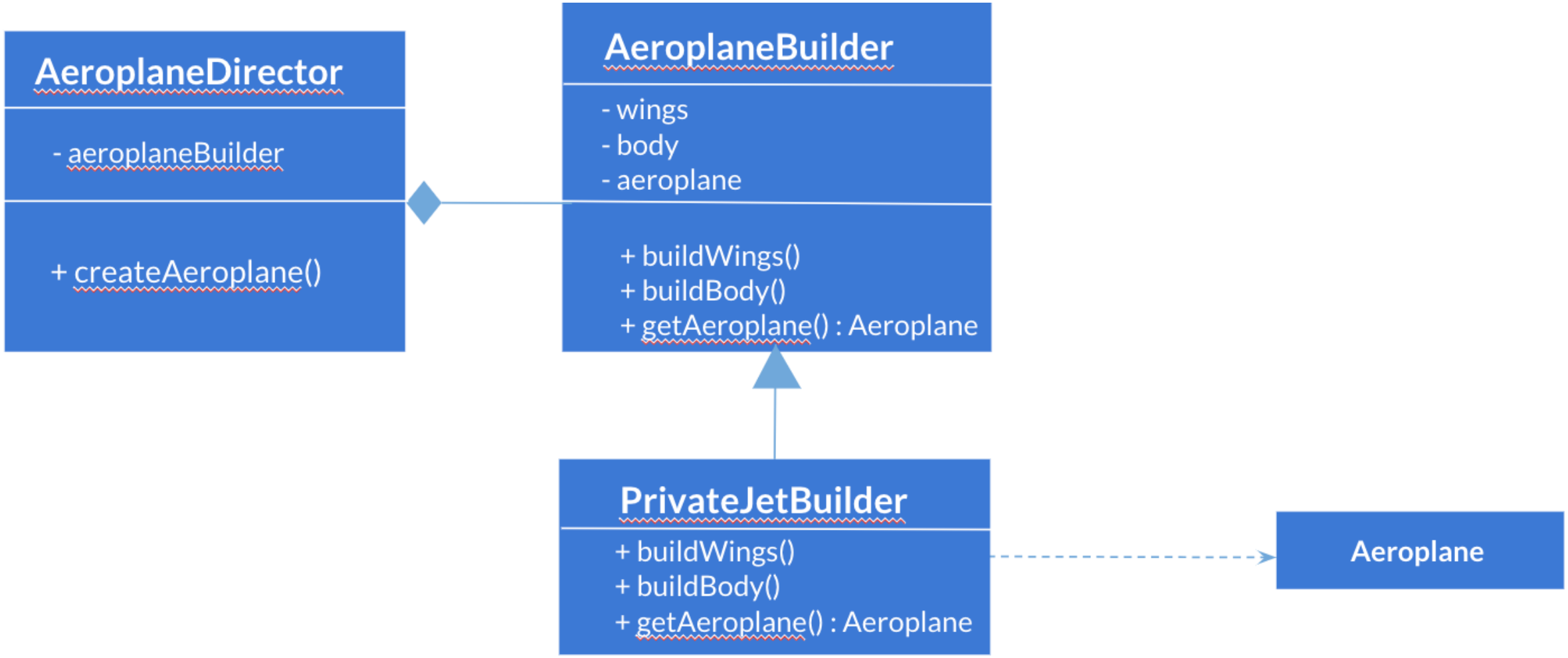
Construction

Builder Pattern

Description

Intent	Motivation	Consequences	Use cases
Ensure class has only ONE instance and provide a global point of access	One instance serving the system Easy access to instance	Controlled access to sole instance Permits refinement of operations and representation	Logger Utility, DB Connection, Account System per company or a printer spool

UML Diagram


AeroplaneDirector
- aeroplaneBuilder
+ createAeroplane()
AeroplaneBuilder
- wings
- body
- aeroplane
+ buildWings()
+ buildBody()
+ getAeroplane() : Aeroplane
PrivateJetBuilder
+ buildWings()
+ buildBody()
+ getAeroplane() : Aeroplane
Aeroplane

Implementation

```
abstract class AeroplaneBuilder {
    abstract void buildWings();
    abstract void buildBody();
    abstract Aeroplane getAeroplane();
 }

class PrivateJetBuilder extends
AeroplaneBuilder {
   private Aeroplane aeroplane;
   public void buildWings() {}
   public void buildBody() {}
   public Aeroplane getAeroplane() {
      return this.aeroplane; }
 }
}
```

```
class BoingtBuilder extends AeroplaneBuilder {
   public void buildWings() {}
   public void buildBody() {}
 }

class AeroplaneDirector {
    public AeroPlane construct(AeroplaneBuilder
aeroplaneBuilder) {
   builder.buildWings();
   builder.buildBody();
   return builder.getAeroplane();
}

AeroplaneBuilder aeroplaneBuilder = new AeroplaneBuilder();
AeroplaneDirector aeroplaneDirector = new
AeroplaneDirector(aeroplaneBuilder);
return aeroplaneDirector.createAeroplane();
```

Construction

Factory Pattern

Description

Intent

Define interface for creating objects

Motivation

Allow subclasses to decide which class to instantiate

Deferred instantiation dynamically at runtime

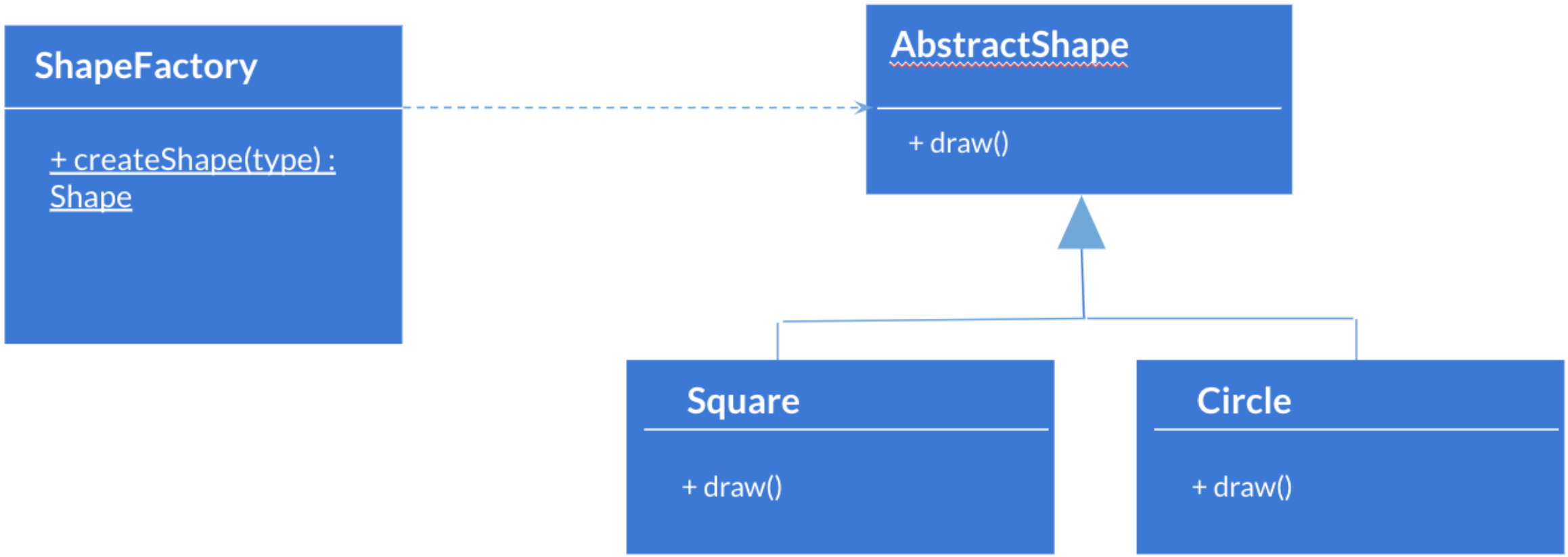
Consequences

Subclasses in control
Connects parallel classes hierarchies

Use cases

java.sql.DriverManager #getConnection()
java.lang.Class#forName()

UML Diagram


ShapeFactory
+ createShape(type) : Shape
AbstractShape
+ draw()
Square
+ draw()
Circle
+ draw()

Implementation

```
class Shape(object):
    # Create based on class name:
    def factory(type):
        if type == "Circle": return Circle()
        if type == "Square": return Square()
        assert 0, "Bad shape creation: " + type
    factory = staticmethod(factory)

class Circle(Shape):
    def draw(self): print("Circle.draw")
    def erase(self): print("Circle.erase")

class Square(Shape):
    def draw(self): print("Square.draw")
      def erase(self): print("Square.erase")
```

Construction

Prototype Pattern

Description

Intent

Define interface for creating objects using a prototype and new objects clone this prototype

Motivation

System is independent of how its products are created
Classes are specified at runtime

Avoids using a hierarchy of factories

Consequences

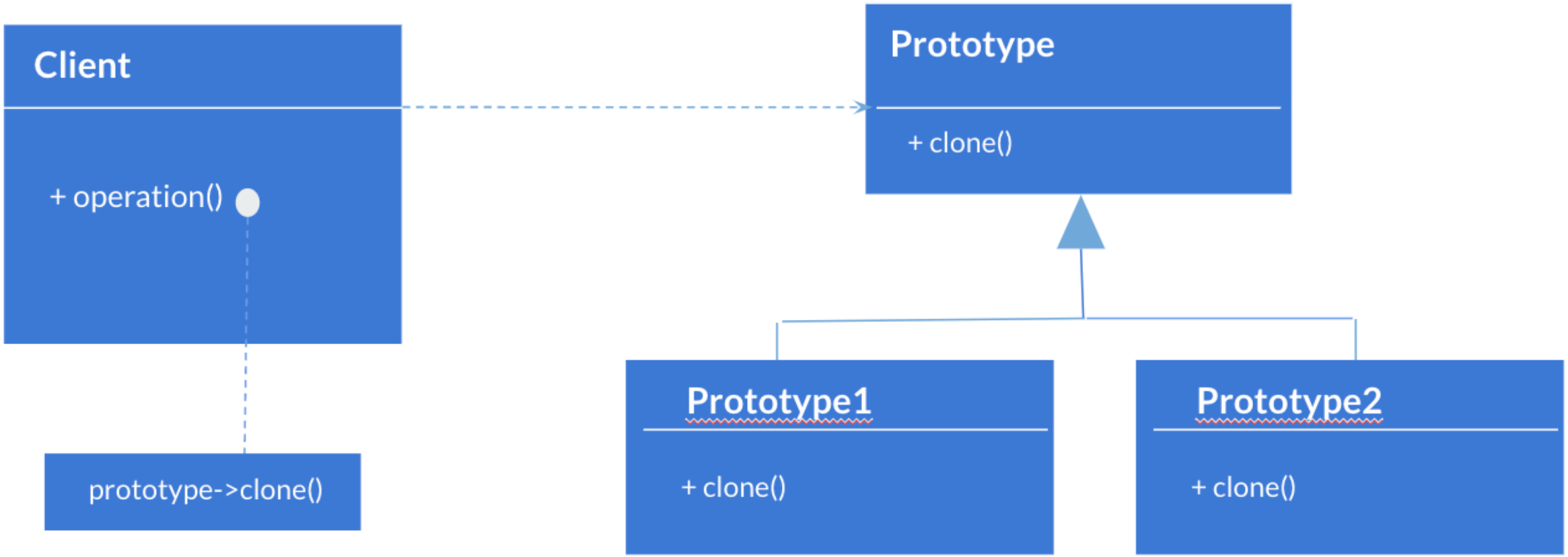
Uses composition instead of inheritance
Reduced subclasses

All subclasses must implement clone()

Use cases

Java out of the box with a Cloneable interface

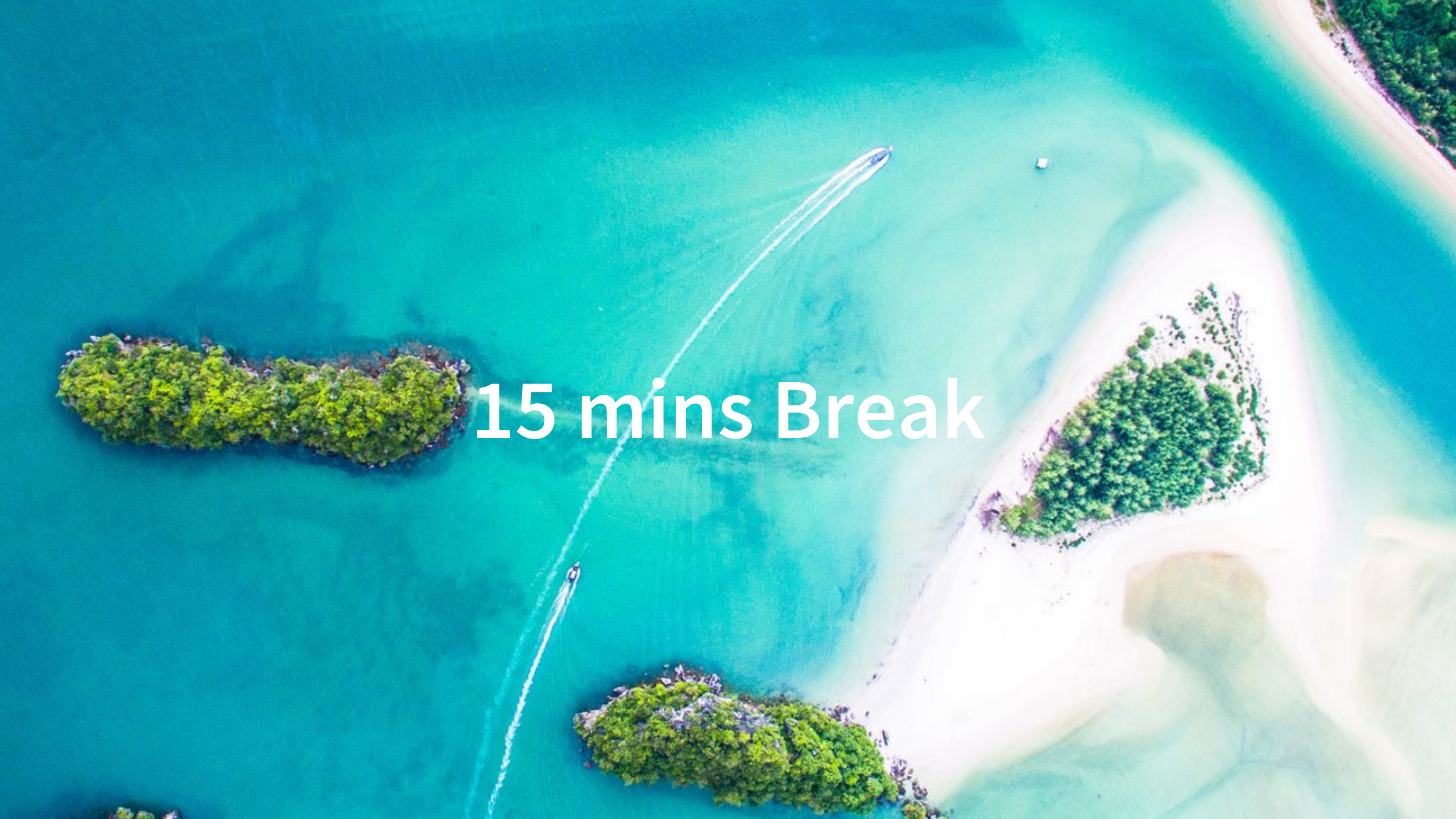
UML Diagram


Client
+ operation()
prototype->clone()
Prototype
+ clone()
Prototype1
+ clone()
Prototype2
+ clone()

Implementation

```
class AeroplanePrototypeFactory extends AeroplaneFactory {
 private Wings prototypeWings;
 private Body prototypeBody;

 public void AeroplanePrototypeFactory(Wings prototypeWings,
Body prototypeBody) {
      this.prototypeWings = prototypeWings;
      this.prototypeBody = prototypeBody;
 }

 public Wings createWings() {
   return prototypeWings.clone();
 }

 public Body createBody() {
   return prototypeBody.clone();
 }
}
```

```
class SmallWings extends Wings {}
class SmallBody extends Body {}

AeroplanePrototypeFactory
aeroplanePrototypeFactory1 = new
AeroplanePrototypeFactory(new Wings(), new
Body());

AeroplanePrototypeFactory
aeroplanePrototypeFactory2 = new
AeroplanePrototypeFactory (new SmallWings(),
new SmallBody());

Aeroplane aeroplane;
aeroplane.createAeroplane(aeroplanePrototype
Factory1);
aeroplane.createAeroplane(aeroplanePrototype
Factory2);
```


15 mins Break

Exercises

Account Management

1. Step 1: git repo clone session three https://github.com/SMostaf/COMPSCIX418.2Step
2. Check the README file
3. Deliver classes
4. Deliver UML Diagram

Code Review

Singlegton Pattern

1. Step 1: git repo clone session three https://github.com/SMostaf/COMPSCIX418.2Step
2. Check the code review folder
3. Step 3: discuss difference between creational methods for the Singleton pattern

Code Refactor

1. Step 1: git repo clone session three https://github.com/SMostaf/COMPSCIX418.2Step
2. Check the README file
3. Discuss what needs refactoring
4. Deliver refactored classes

Error Handling

1. Step 1: git repo clone session three https://github.com/SMostaf/COMPSCIX418.2Step
2. Check the README file
3. Discuss ways to improve error handling
4. Deliver refactored classes

Next Week

Session Four

1. **Review Session Three**
2. **Structural Design Patterns**
   Adaptor- Decorator- Facade- Proxy
3. **In class exercises**