CS 525 - ASD

Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

Advanced Software Development

Maharishi University
OF MANAGEMENT

Lesson 11 Framework implementation

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

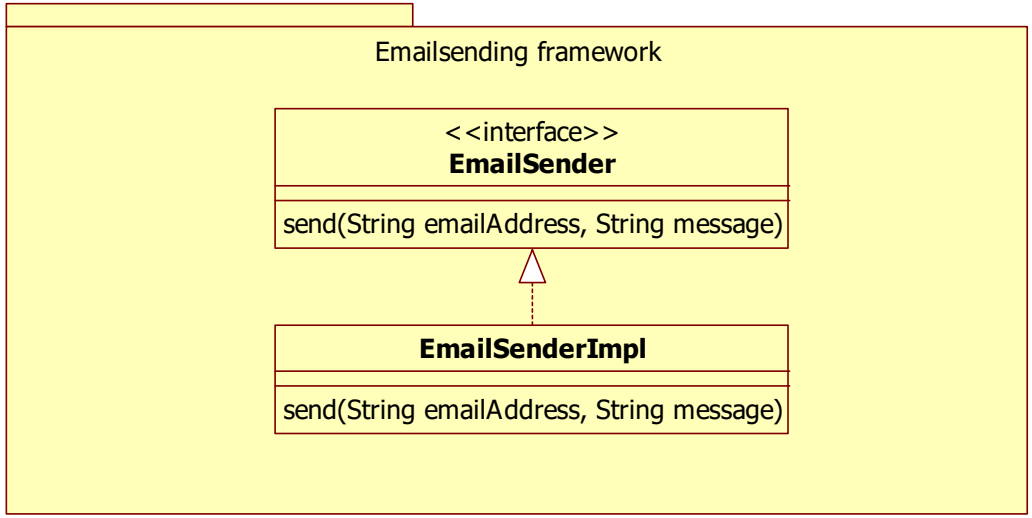
L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

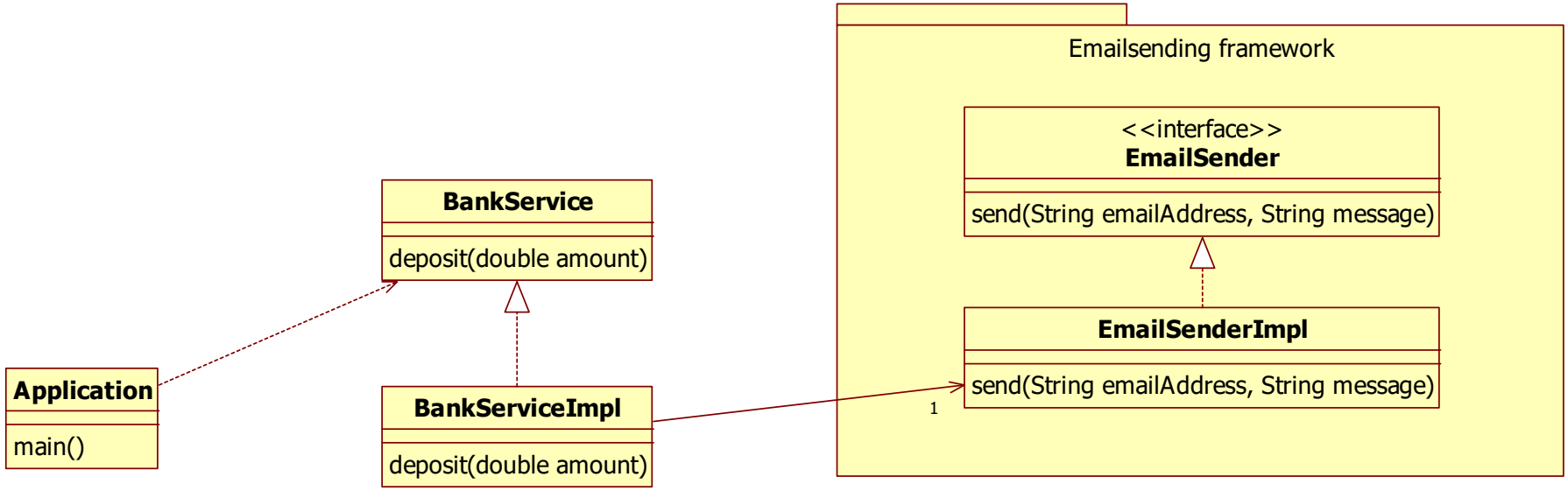
Simple framework

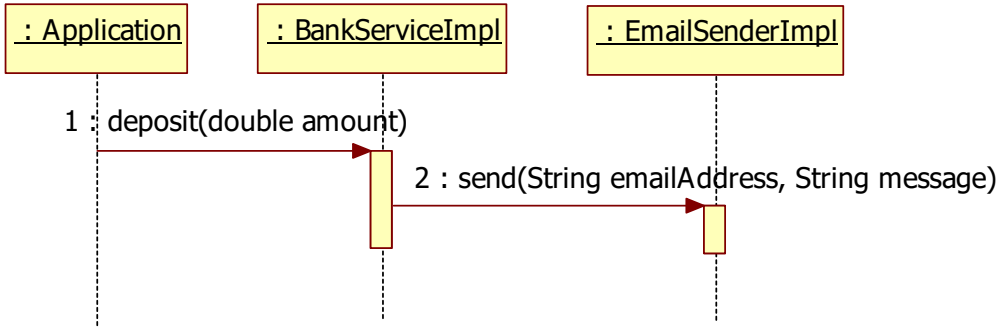
```
public interface EmailSender {
  void send(String emailAddress, String message);
}
```

```
public class EmailSenderImpl implements EmailSender {

  public void send(String emailAddress, String message) {
    System.out.println("sending email to "+emailAddress+ " , message="+message);
  }
}
```



Using the framework





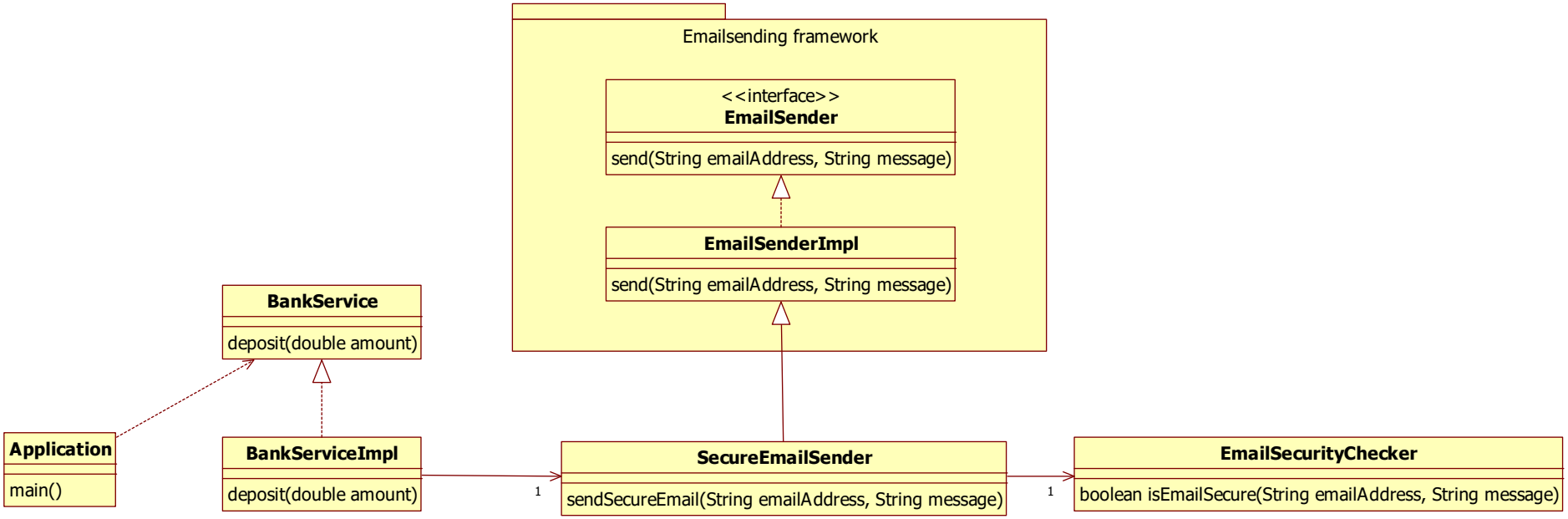
Using the framework

```
public class Application {

  public static void main(String[] args)  {
    BankService bankService = new BankServiceImpl();
    EmailSender emailSender = new EmailSenderImpl();
    bankService.setEmailSender(emailSender);

    bankService.deposit(100.0);
  }
}
```

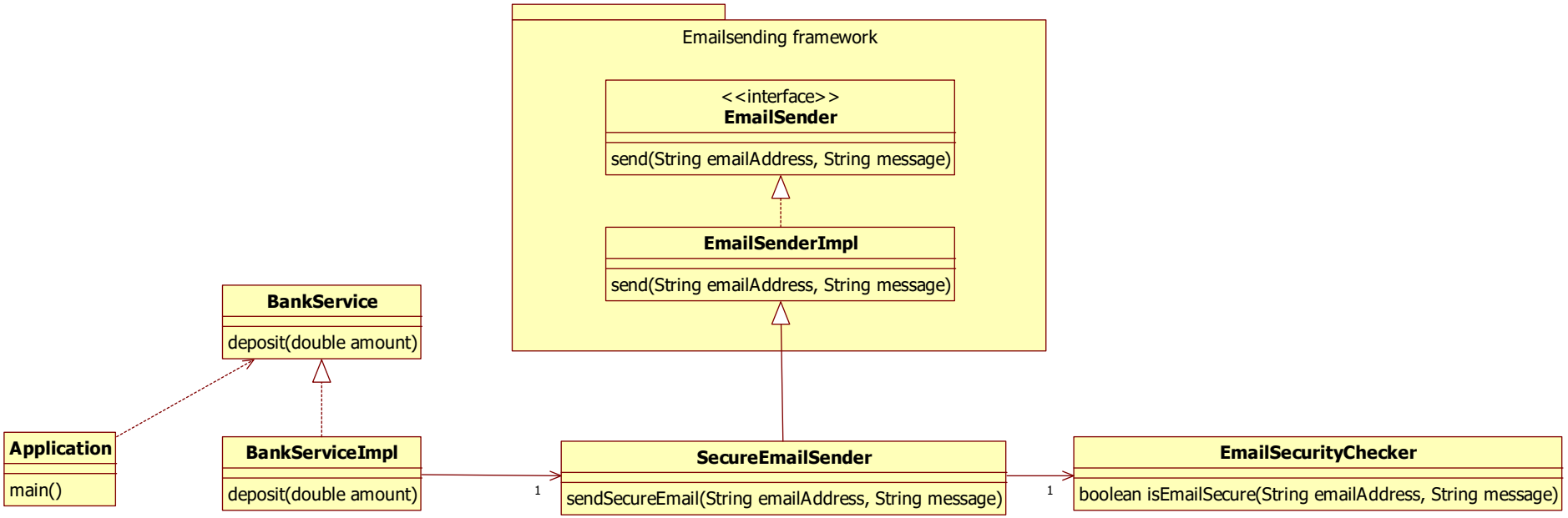
```
public interface BankService {
  void deposit(double amount);
  void setEmailSender(EmailSender emailService);
}
```

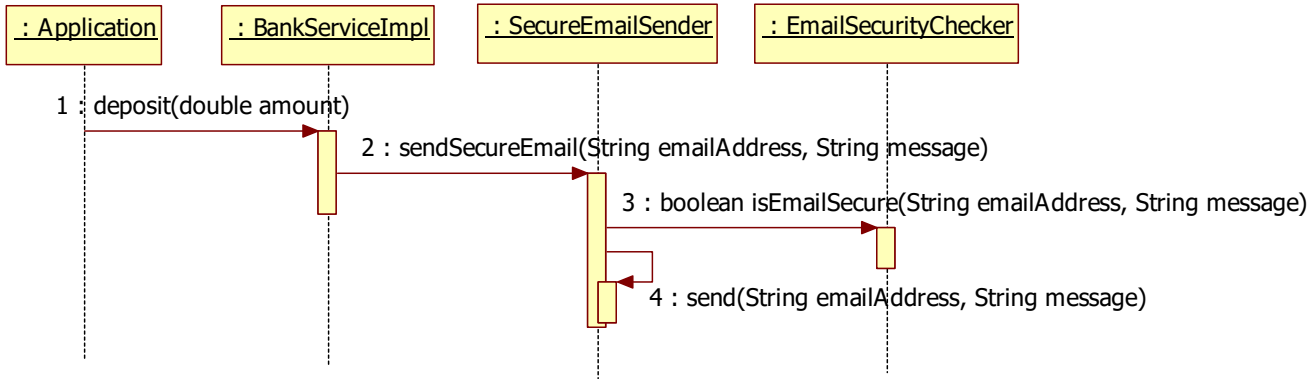
```
public class BankServiceImpl implements BankService {
  private EmailSender emailSender;

  public void setEmailSender(EmailSender emailSender) {
    this.emailSender = emailSender;
  }

  public void deposit(double amount) {
    emailSender.send("customer@gmail.com", "deposit of $" + amount);
  }
}
```

Using the framework



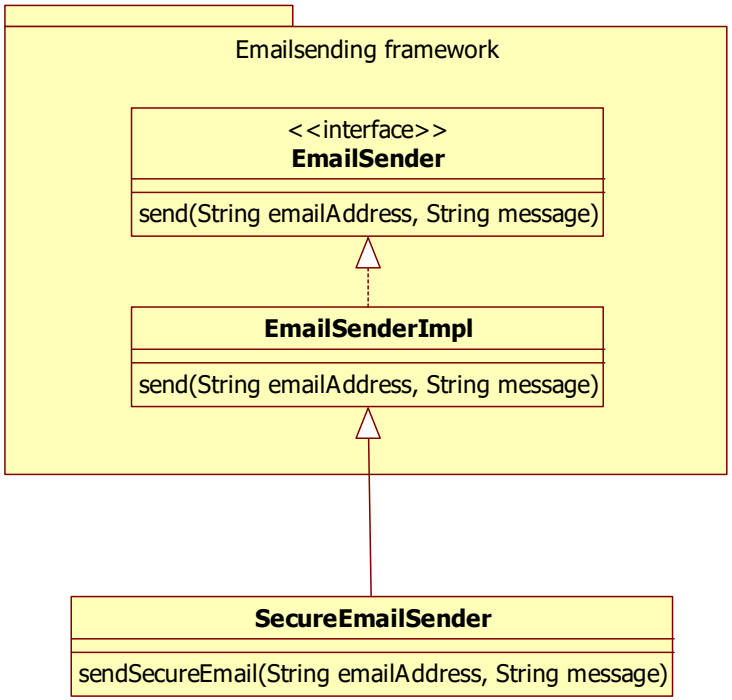
Inheritance



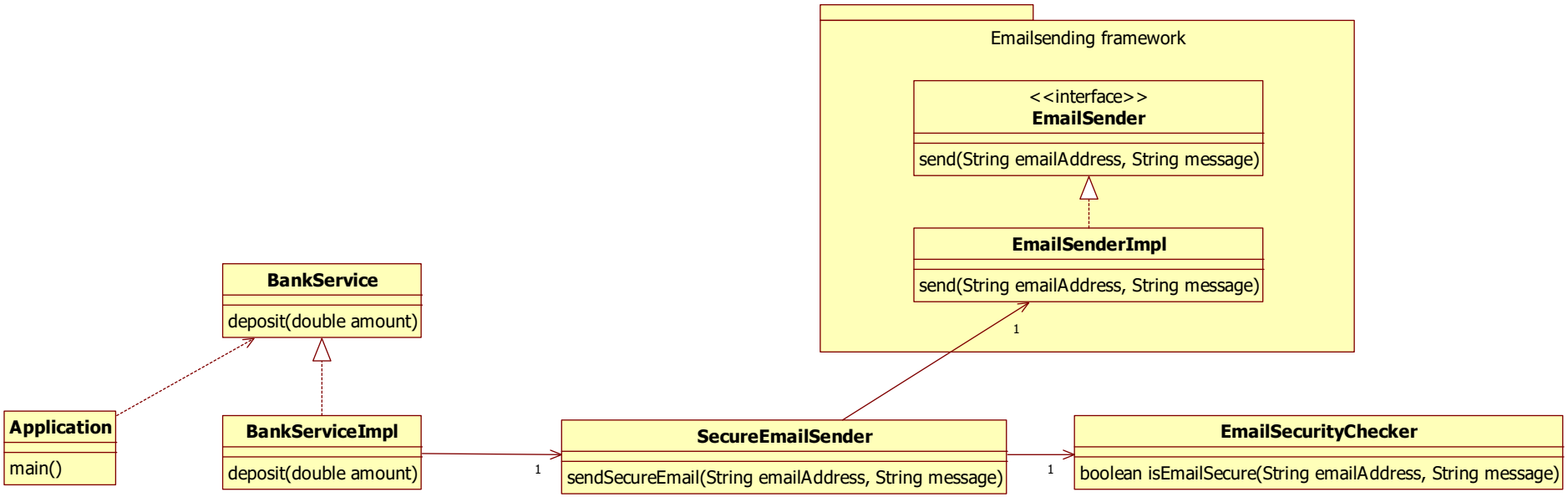


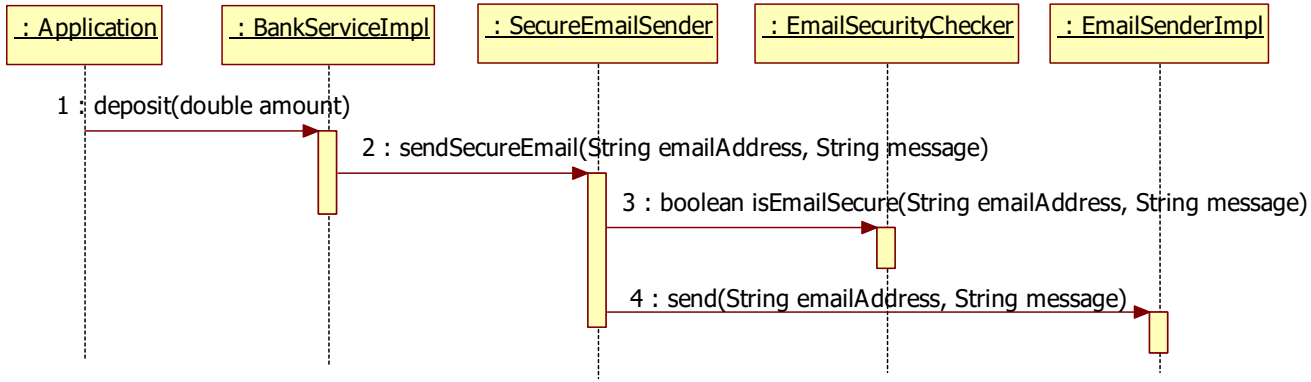
Inheritance

```
public class SecureEmailSender extends EmailSenderImpl{
  EmailSecurityChecker emailSecurityChecker = new EmailSecurityChecker();

  public void sendSecureEmail(String emailAddress, String message) {
    if (emailSecurityChecker.isEmailSecure(emailAddress, message)) {
      send(emailAddress, message);
    }
  }
}
```



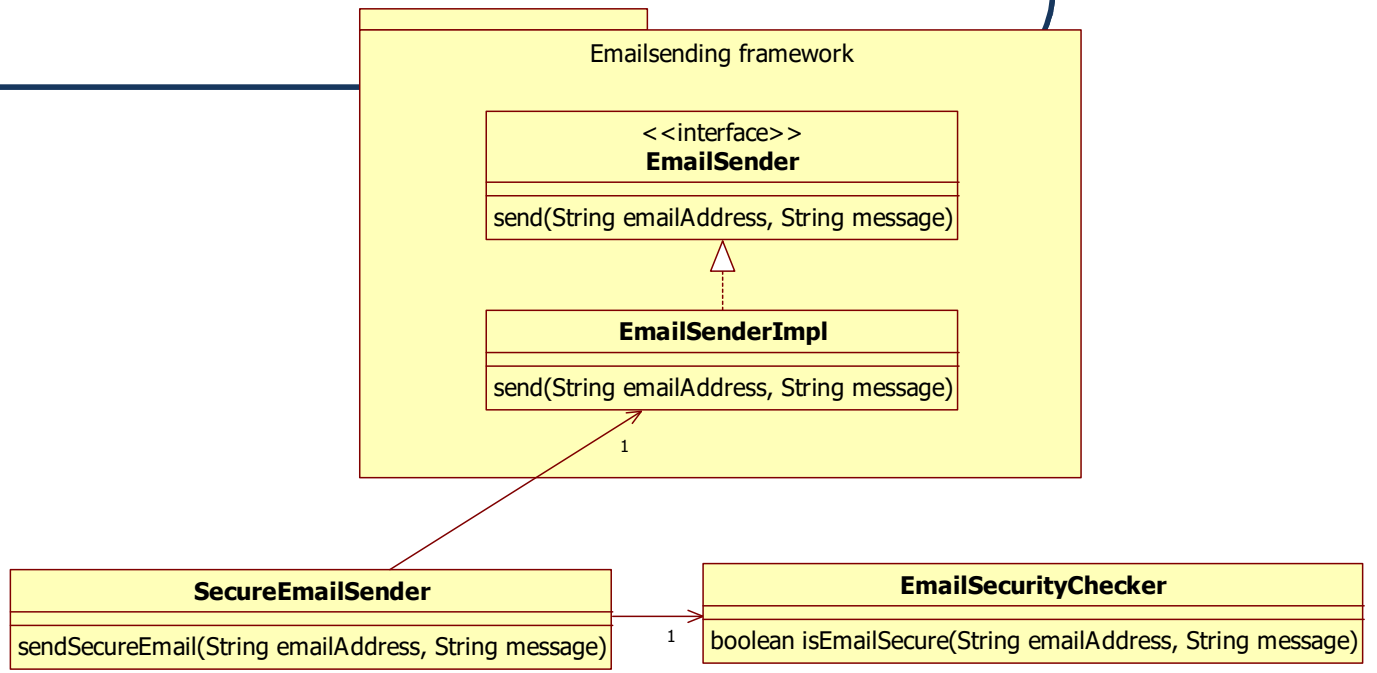
Composition





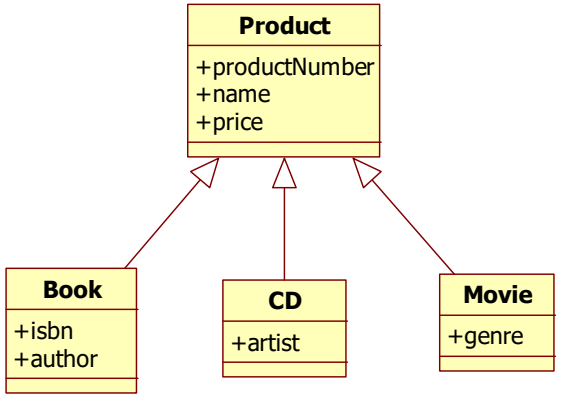
Composition

```
public class SecureEmailSender {
  EmailSecurityChecker emailSecurityChecker = new EmailSecurityChecker();
  EmailSender emailSender;

  public void setEmailSender(EmailSender emailSender) {
    this.emailSender = emailSender;
  }

  public void sendSecureEmail(String emailAddress, String message) {
    if (emailSecurityChecker.isEmailSecure(emailAddress, message)) {
      emailSender.send(emailAddress, message);
    }
  }
}
```

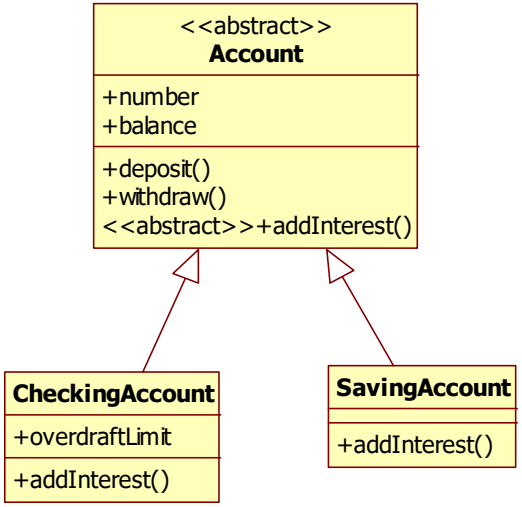


Advantages of inheritance

- Code reusability
  - Minimize the amount of duplicate code



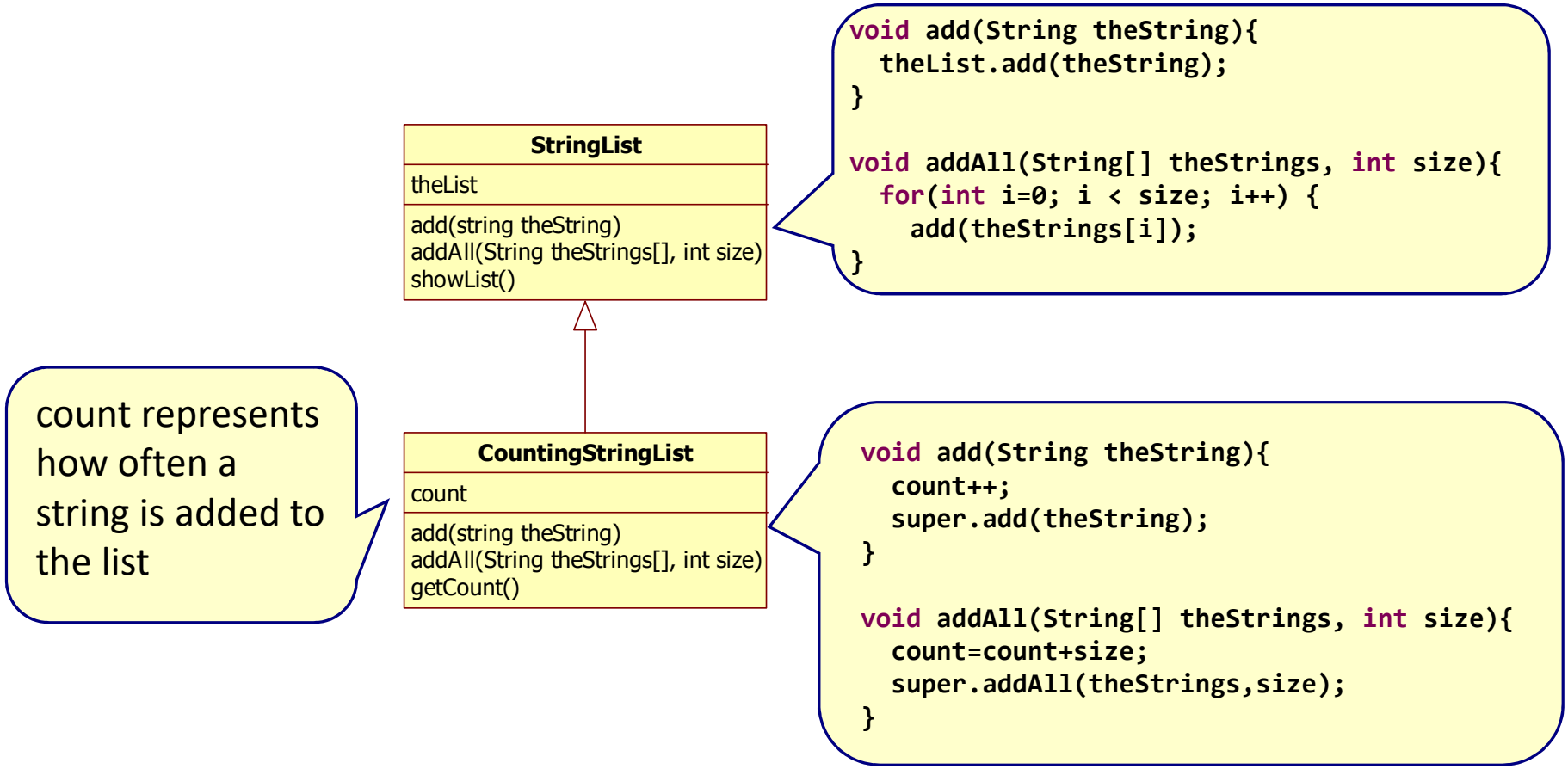
- Code flexibility
  - Classes that inherit from a common superclass can be used interchangeably (polymorphism)



Disadvantages of inheritance

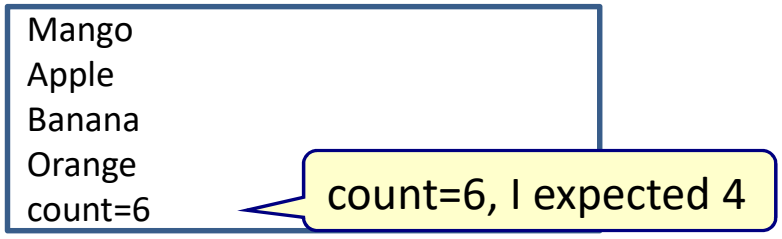
- The base class and sub class are tightly coupled
  - Every change in the base class ripples down to the subclasses
  - Subclasses are entirely dependent on their superclass
    - If you write the subclass you need to understand the base class
      - Breaks encapsulation
- Multiple inheritance problem

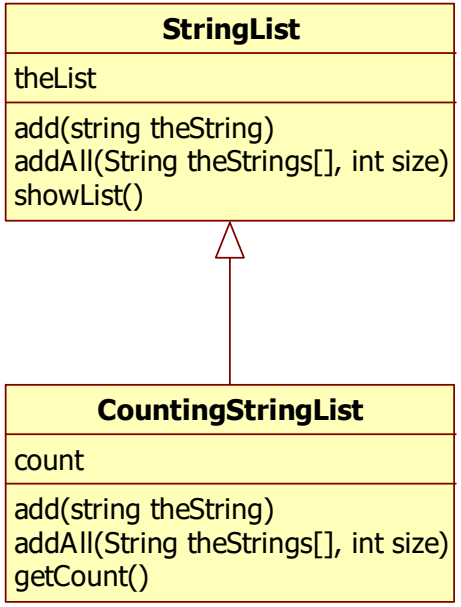
Inheritance problem 1



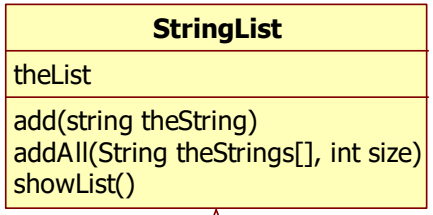
Inheritance problem 1

```
int main(…) {
  CountingStringList list = new CountingStringList();
  string s= "Mango";
  list.add(s);
  string s2= "Apple";
  list.add(s2);
  String[] fruit = {"Banana", "Orange"};
  list.addAll(fruit,2);
  list.showList();
  System.out.println("count="+list.getCount());
}
```

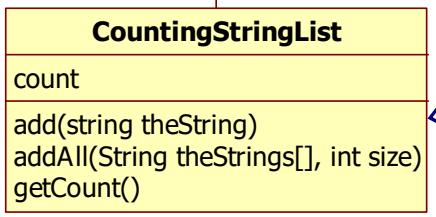




What is the problem?

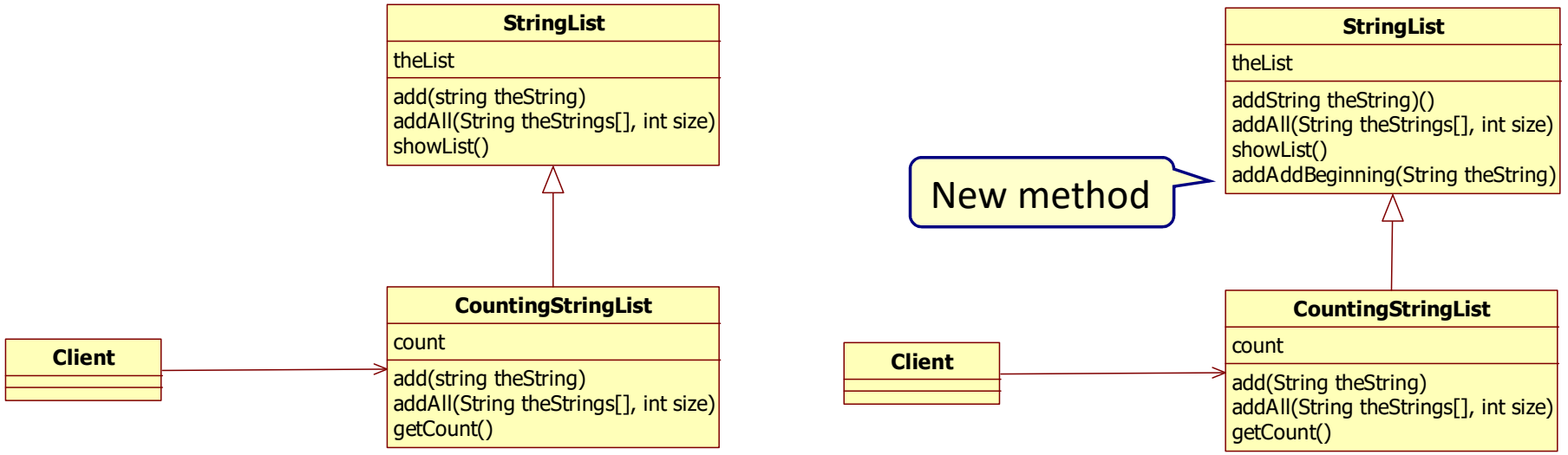



```
void add(string theString){
  theList.add(theString);
}

void addAll(String theStrings[], int size){
  for(int i=0; i < size; i++) {
    add(theStrings[i]);
}
```



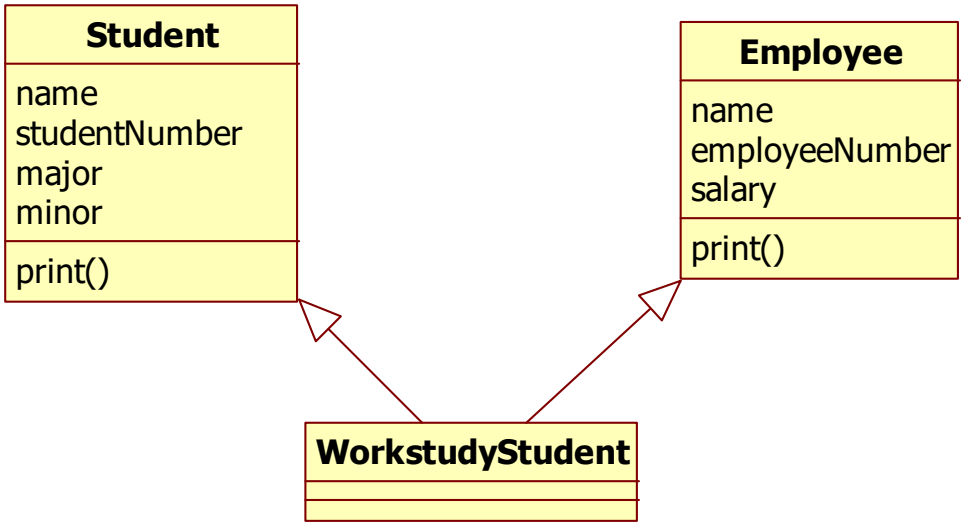

```
void add(string theString){
  count++;
  super.add(theString);
}

void addAll(String theStrings[], int size){
  count=count+size;
  super.addAll(theStrings,size);
}
```

Inheritance problem 2



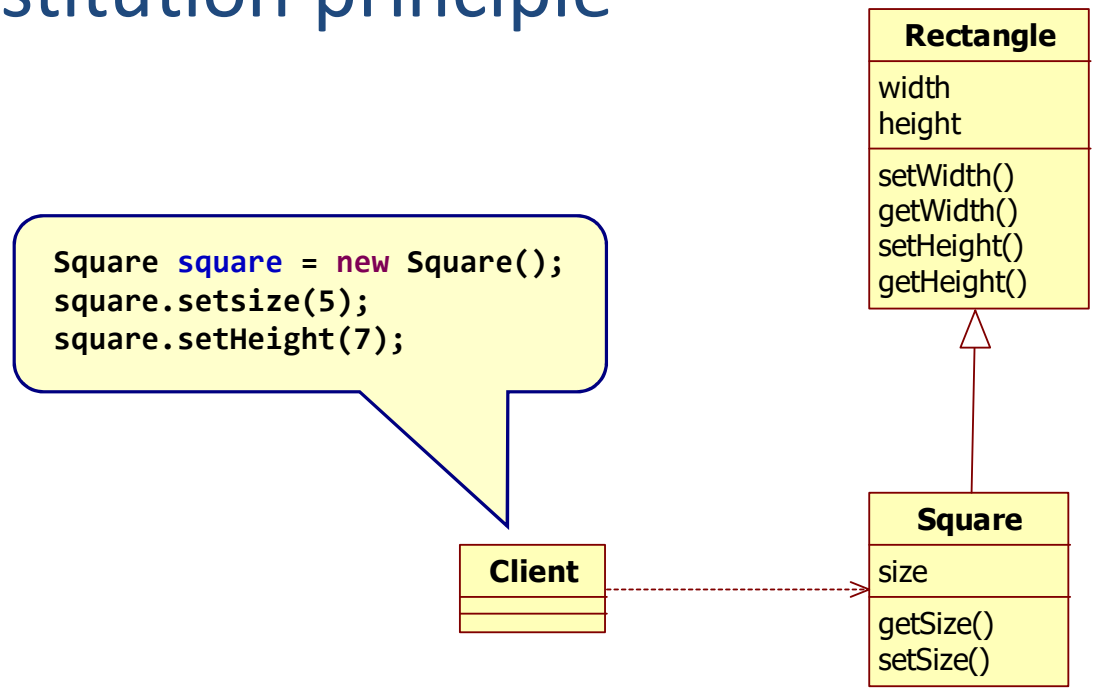
Inheritance problem 3

- Multiple inheritance is not allowed in Java



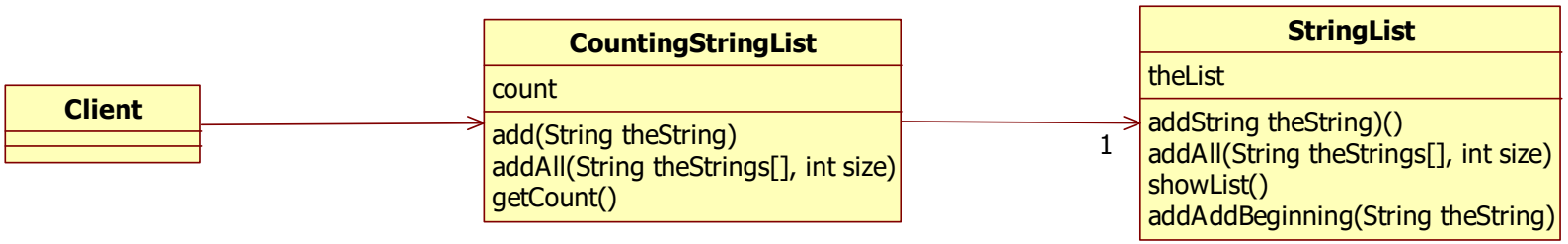
Inheritance problem 4

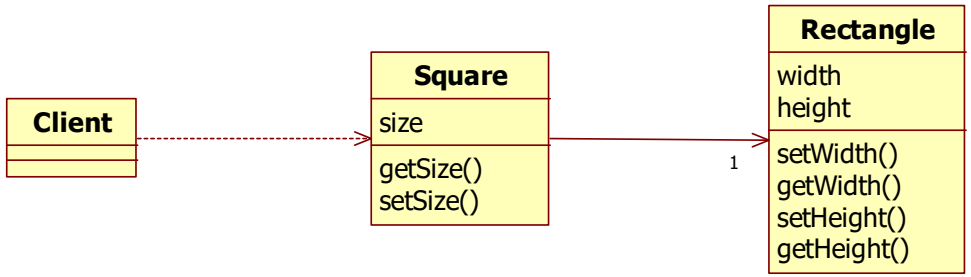
- Liskov substitution principle



Solution to all inheritance problems

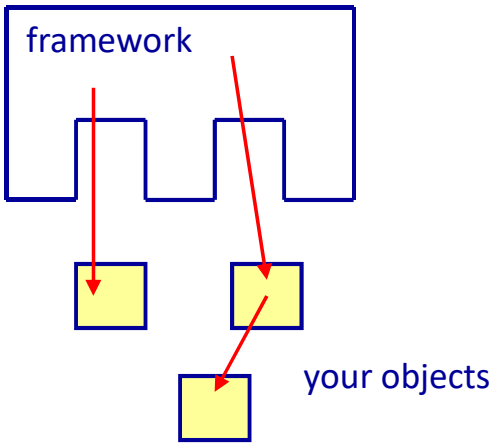
- Favor composition over inheritance





INVERSION OF CONTROL

Inversion of Control (IoC)

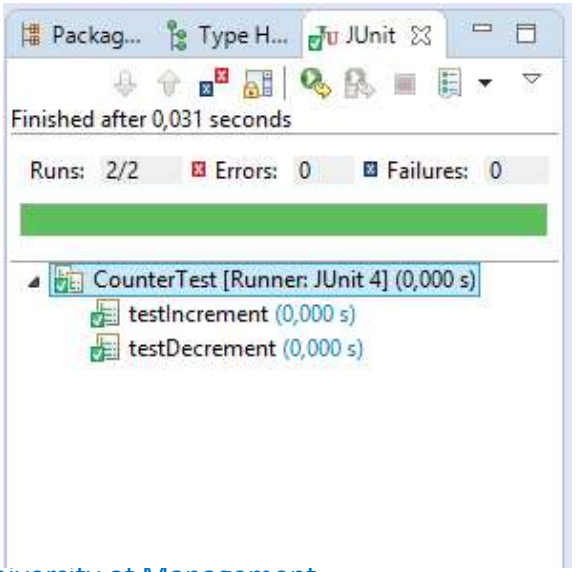


IoC: The framework calls your code

Example of JUnit framework

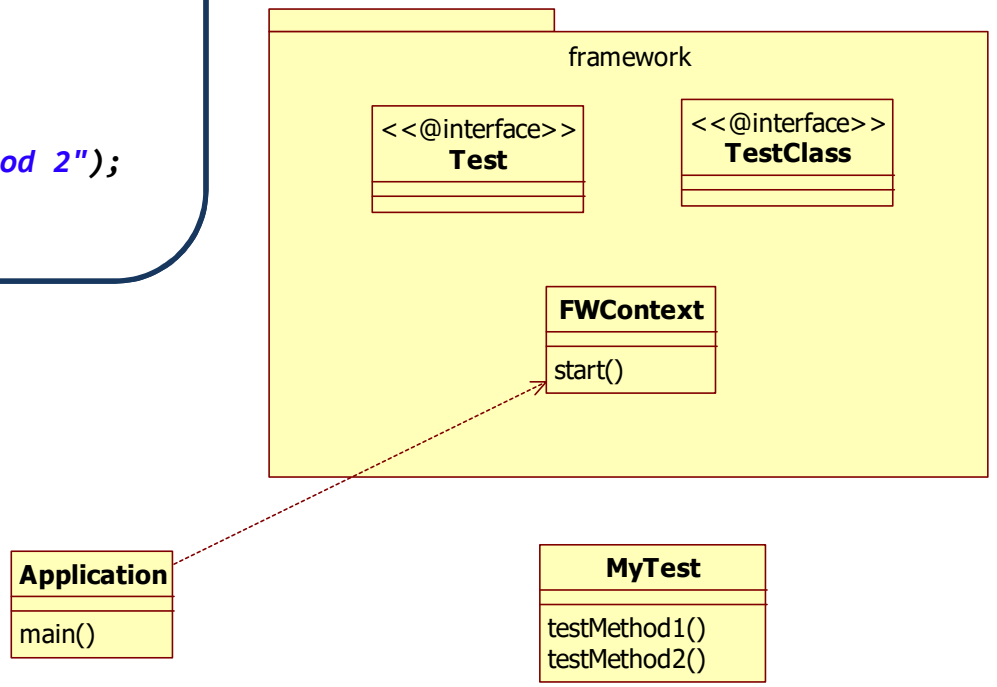
```
public class CounterTest {
      @Test
      public void testIncrement(){
         Counter counter = new Counter();
         assertEquals(1,counter.increment());
         assertEquals(2,counter.increment());
      }

      @Test
      public void testDecrement(){
         Counter counter = new Counter();
         assertEquals(-1,counter.decrement());
         assertEquals(-2,counter.decrement());
      }
}
```

```
public class Counter {
     private int counterValue=0;

     public int increment(){
        return ++counterValue;
     }
     public int decrement(){
        return --counterValue;
     }
}
```



Inversion of Control framework

```
@TestClass
public class MyTest {

  @Test
  public void testMethod1() {
    System.out.println("perform test method 1");
  }

  @Test
  public void testMethod2() {
    System.out.println("perform test method 2");
  }
}
```

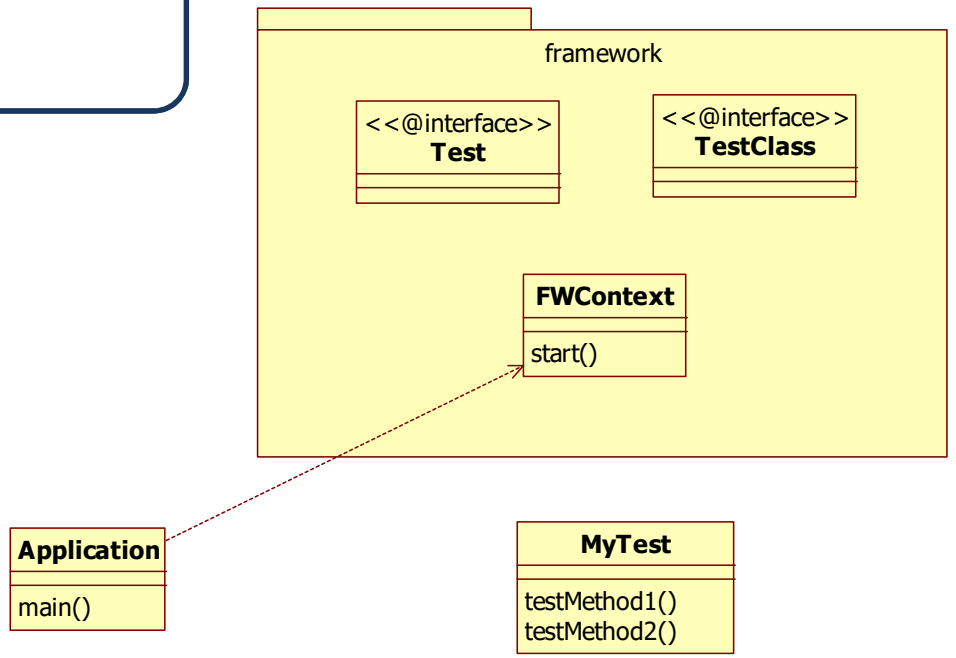


Inversion of Control framework

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Create your own annotations in Java

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface TestClass {
}
```



Define your own annotations

Retention: defines the visibility of the annotation

- SOURCE—Annotation is visible only at the source level and will be ignored by the compiler.
- CLASS—Annotation is visible by the compiler at compile time, but will be ignored by the VM.
- RUNTIME—Annotation is visible by the VM so they can be read only at run-time.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Target: where can I apply this annotation?
@Target(value={TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})

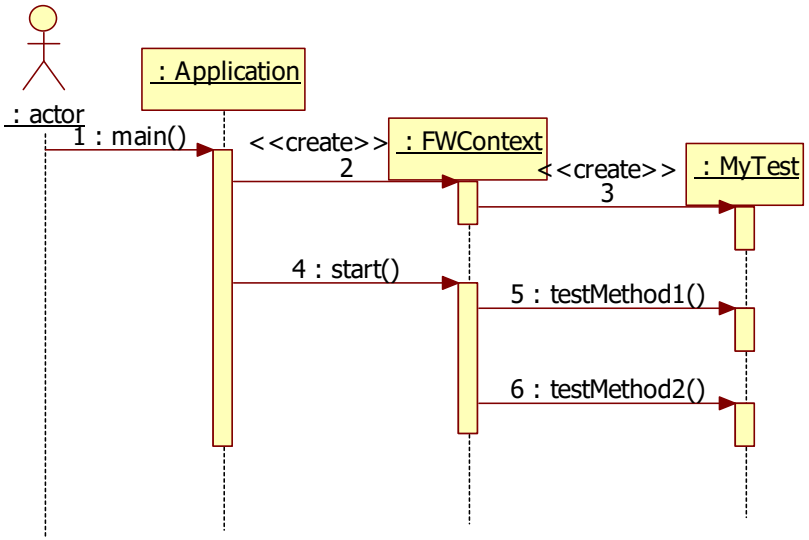
Classpath scanning

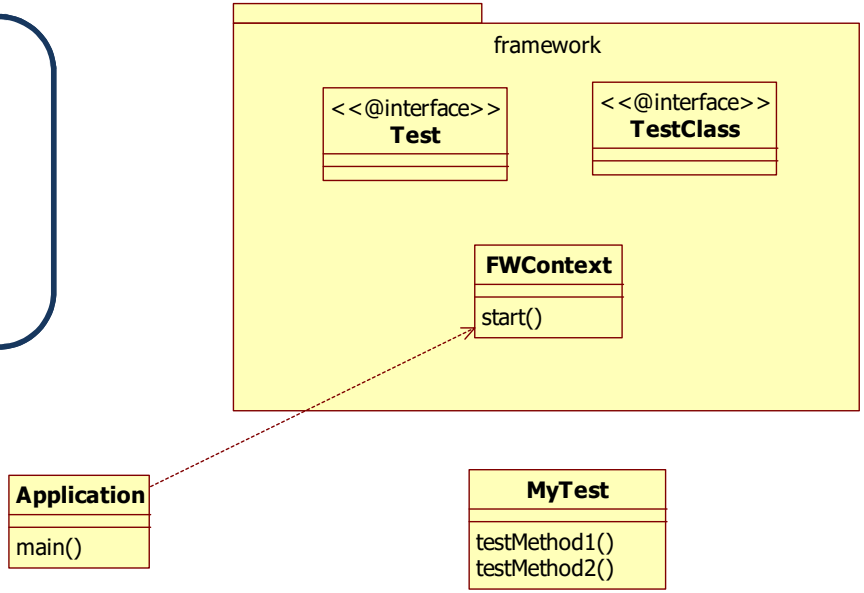
```
public class FWContext {
  private static List<Object> objectMap = new ArrayList<>();

  public FWContext() {
    try {
      // find and instantiate all classes annotated with the @TestClass annotation
      Reflections reflections = new Reflections("");
      Set<Class<?>> types = reflections.getTypesAnnotatedWith(TestClass.class);
      for (Class<?> implementationClass : types) {
        objectMap.add((Object) implementationClass.newInstance());
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  public void start() {
    try {
      for (Object theTestClass : objectMap) {
        // find all methods annotated with the @Test annotation
        for (Method method : theTestClass.getClass().getDeclaredMethods()) {
          if (method.isAnnotationPresent(Test.class)) {
            method.invoke(theTestClass);
          }
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

Classpath scanning

Inversion of Control framework

```
public class Application {

  public static void main(String[] args) {
    FWContext fWContext = new FWContext();
    fWContext.start();
  }
}
```






```
perform test method 2
perform test method 1
```

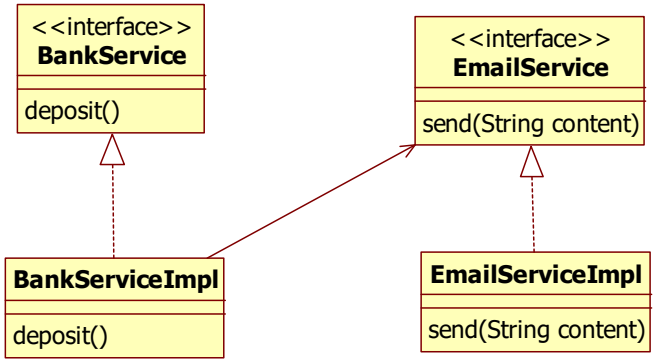
DEPENDENCY INJECTION

Instantiate an object directly

```
public interface BankService {
  public void deposit() ;
}
```

```
public class BankServiceImpl implements BankService{
  private EmailService emailService= new EmailServiceImpl();

  public void deposit() {
    emailService.send("deposit");
  }
}
```

Instantiate the EmailService




```
public interface EmailService {
  void send(String content);
}
```

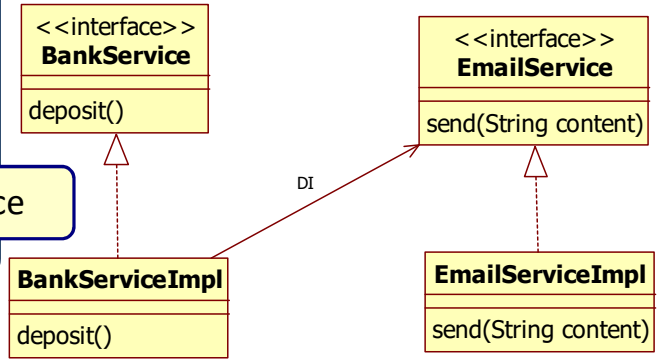
```
public class EmailServiceImpl implements EmailService{

  public void send(String content) {
    System.out.println("sending email: "+content);
  }
}
```

Dependency Injection

```
public interface BankService {
  public void deposit() ;
}
```

```
public class BankServiceImpl implements BankService{
  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void deposit() {
    emailService.send("deposit");
  }
}
```

We can inject any EmailService

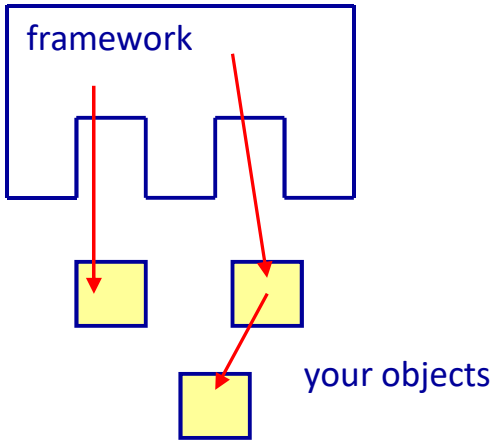



```
public interface EmailService {
  void send(String content);
}
```

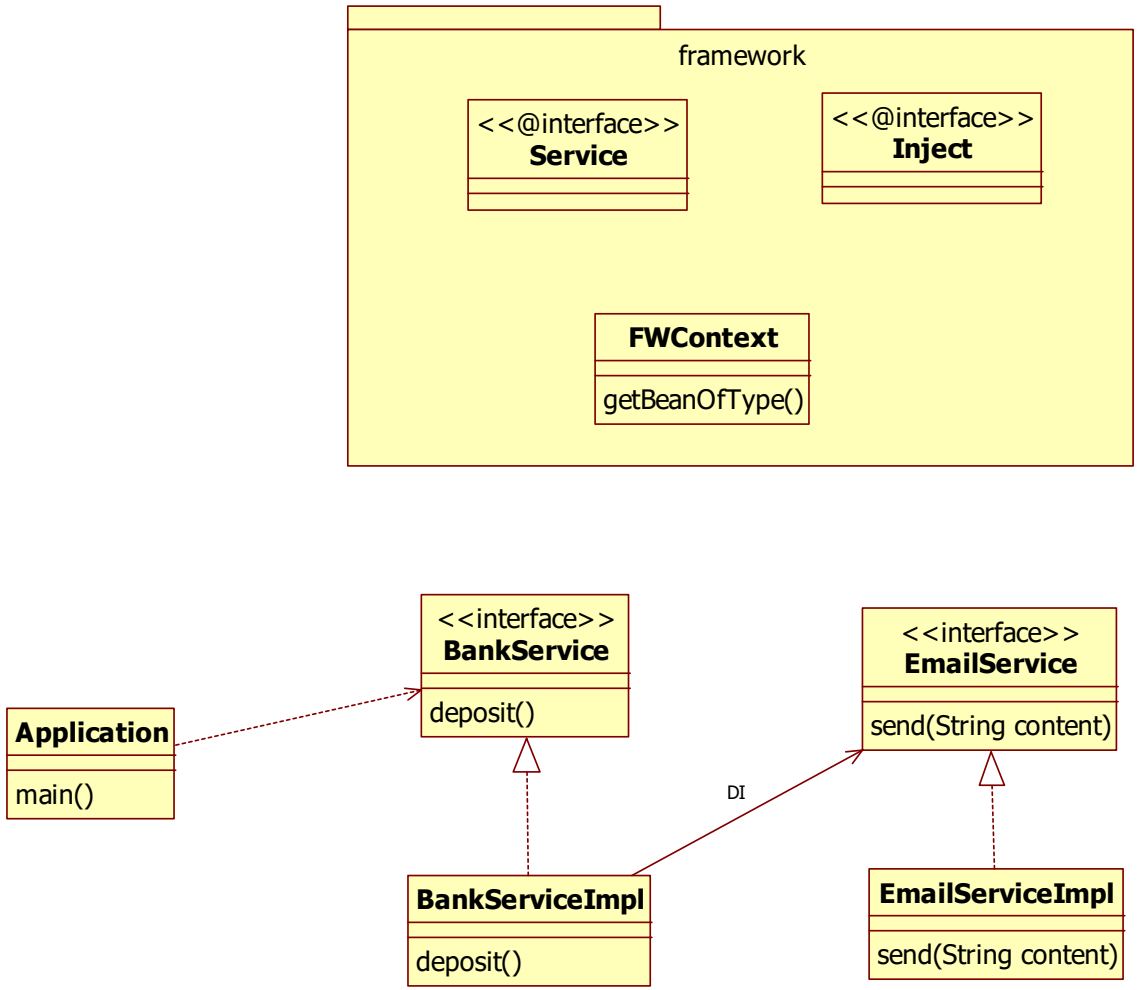
```
public class EmailServiceImpl implements EmailService{

  public void send(String content) {
    System.out.println("sending email: "+content);
  }
}
```

Framework implementation

- IoC: The framework instantiates our application classes
- Dependency injection: The framework wires our objects together



Dependency Injection framework



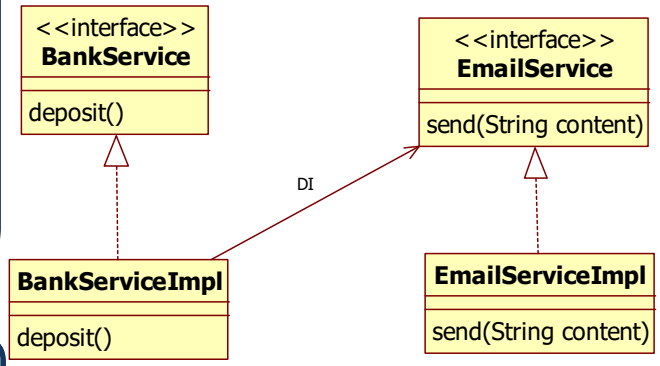
Dependency Injection framework

```java
public interface BankService {
  public void deposit() ;
}
```

```java
@Service
public class BankServiceImpl implements BankService{
  @Inject
  private EmailService emailService;

  public void setEmailService(EmailService emailService) {
    this.emailService = emailService;
  }

  public void deposit() {
    emailService.send("deposit");
  }
}
```

The framework will instantiate this class

The framework will inject the EmailService




```java
public interface EmailService {
  void send(String content);
}
```

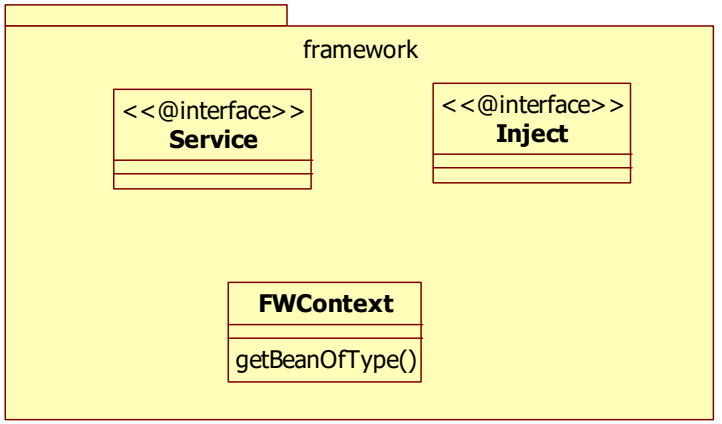
```java
@Service
public class EmailServiceImpl implements EmailService{

  public void send(String content) {
    System.out.println("sending email: "+content);
  }
}
```

The framework will instantiate this class

Dependency Injection framework

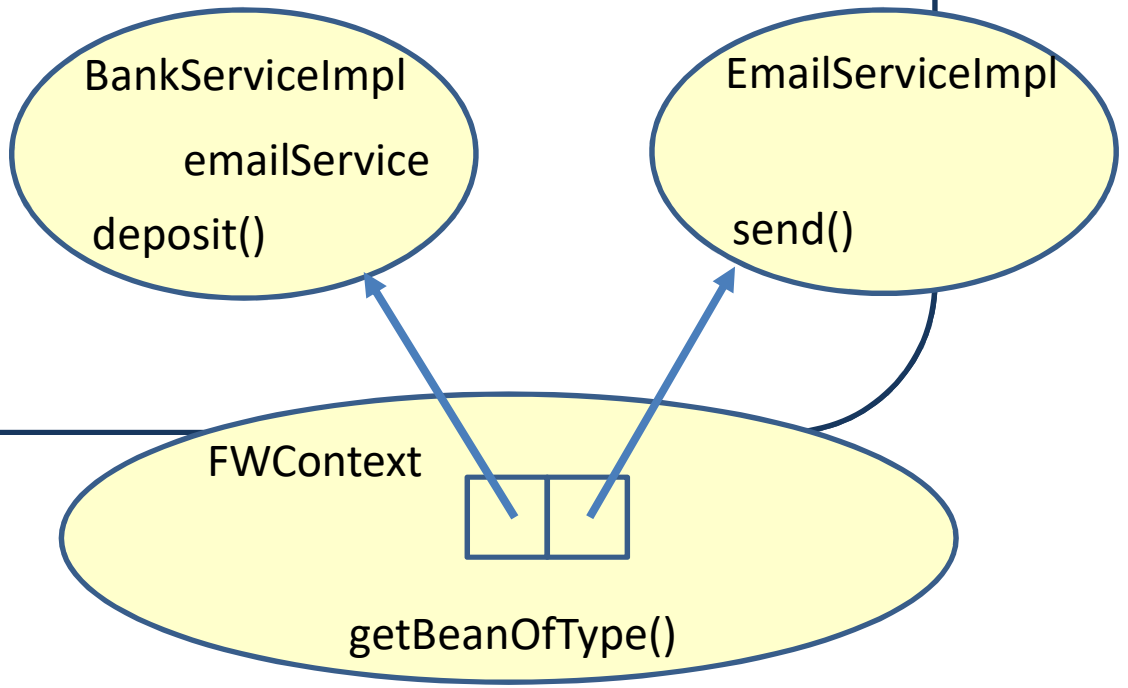
```java
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service {
}
```

```java
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {
}
```



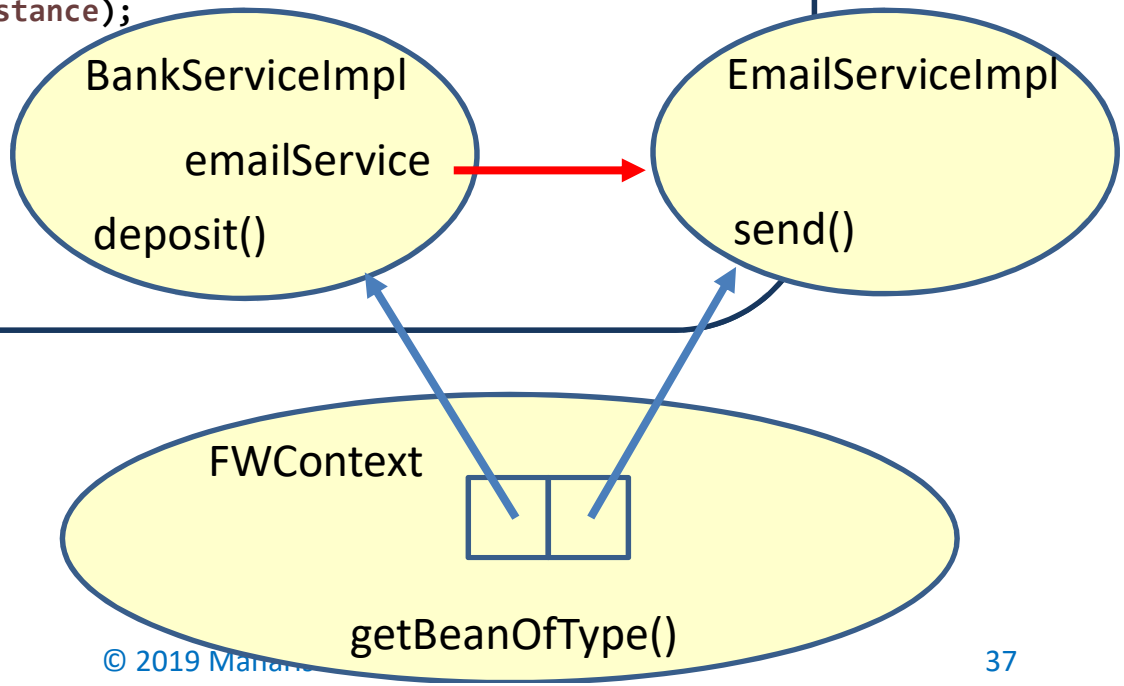
Context class

```
public class FWContext {

  private static List<Object> objectMap = new ArrayList<>();

  public FWContext() {
    try {
    // find and instantiate all classes annotated with the @Service annotation
    Reflections reflections = new Reflections("");
    Set<Class<?>> types = reflections.getTypesAnnotatedWith(Service.class);
    for (Class<?> implementationClass : types) {
      objectMap.add((Object) implementationClass.newInstance());
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  performDI();
}
```



Context class

```
private void performDI() {
  try {
    for (Object theTestClass : objectMap) {
         // find annotated fields
      for (Field field : theTestClass.getClass().getDeclaredFields()) {
        if (field.isAnnotationPresent(Inject.class)) {
            // get the type of the field
            Class<?> theFieldType =field.getType();
            //get the object instance of this type
            Object instance = getBeanOftype(theFieldType);
            //do the injection
            field.setAccessible(true);
            field.set(theTestClass, instance);
         }
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
}
```

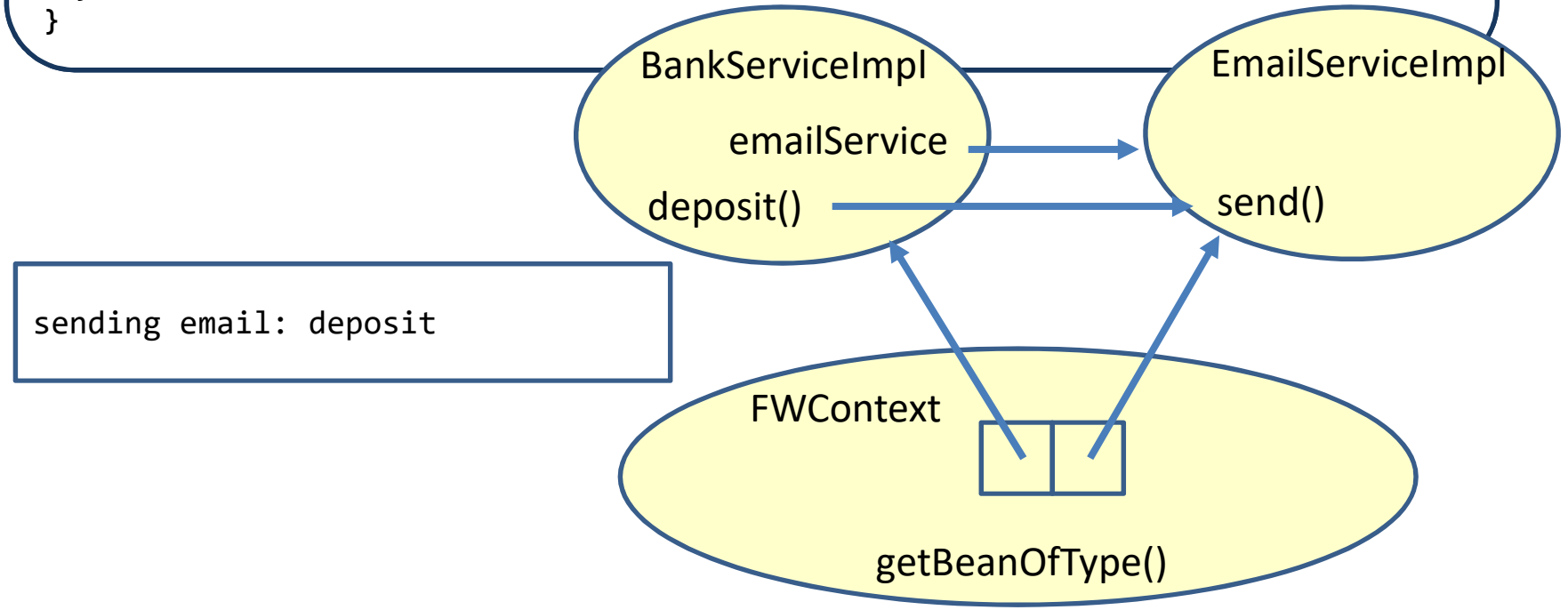


Context class

```
public Object getBeanOftype(Class interfaceClass) {
  Object service = null;
  try {
    for (Object theTestClass : objectMap) {
      Class<?>[] interfaces = theTestClass.getClass().getInterfaces();

      for (Class<?> theInterface : interfaces) {
        if (theInterface.getName().contentEquals(interfaceClass.getName()))
          service = theTestClass;
      }
    }
  } catch (Exception e) {
    e.printStackTrace();
  }
  return service;
}
```

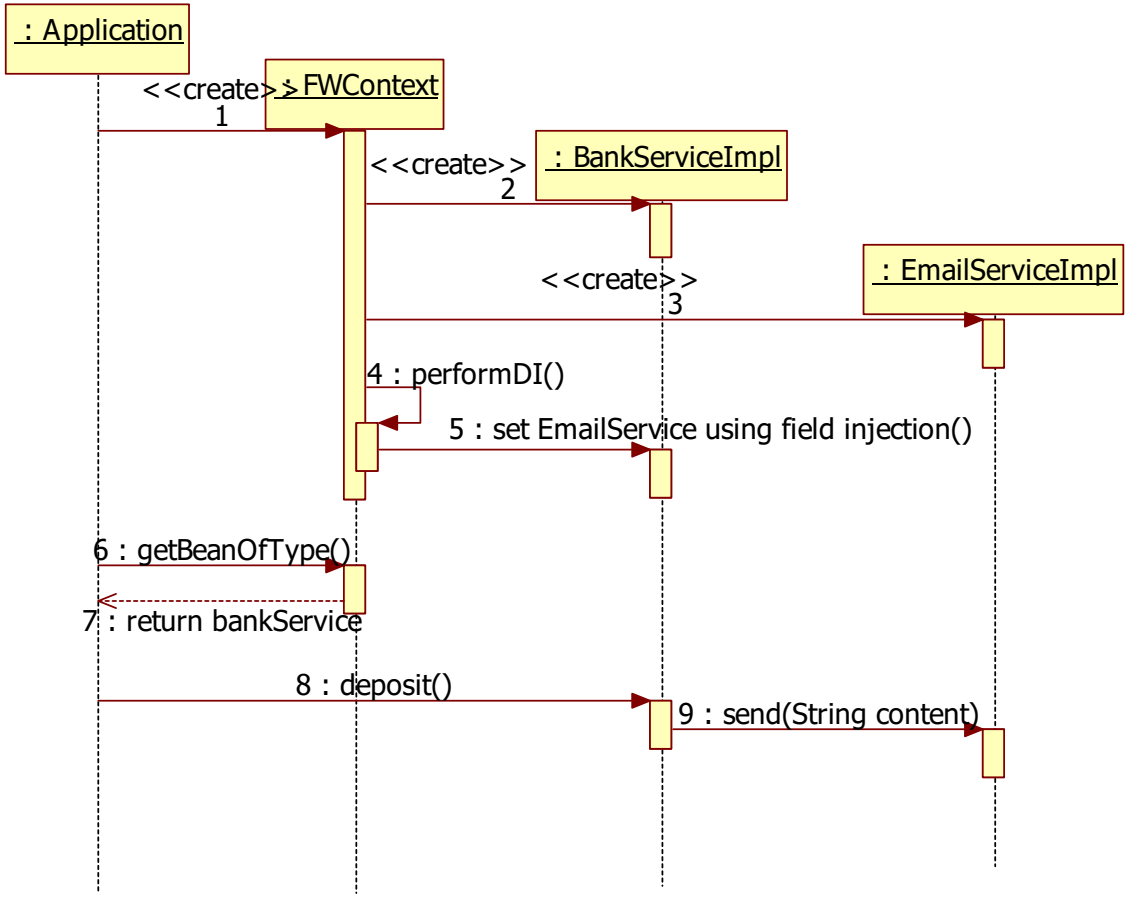
Dependency Injection framework

```
public class Application {

  public static void main(String[] args) {
    FWContext fWContext = new FWContext();

    BankService bankService =  (BankService) fWContext.getBeanOftype(BankService.class);
    if (bankService != null)
       bankService.deposit();
  }
}
```




```
sending email: deposit
```

Dependency Injection framework



Main point

- Dependency injection gives us flexibility in wiring objects together.
- Daily contact with pure consciousness results in more and more happiness by spontaneous right action.