Dependency Management Techniques

Kirk Knoernschild

TeamSoft, Inc.

www.teamsoftinc.com

http://techdistrict.kirkk.com

http://www.kirkk.com

pragkirk@kirkk.com

Agenda

- Introduction
- Class dependencies
- Package dependencies
- Binary dependencies

Summary

- Class Dependencies
  - Inheritance
  - Association
- Package Dependencies
  - Acyclic Dependencies
  - Testing with Jdepend
- Binary Dependencies
  - Inverting Relationships
  - Callback, Escalation, Demotion
  - JarAnalyzer

Introduction

- Some dependencies are necessary
- Too many dependencies cause problems
  - Hinder maintenance
  - Prevent extensibility
  - Inhibit reusability
  - Restrict testability
- Managing dependencies is about minimizing coupling.

Types of Dependencies

- Class dependencies
  - Relationships between classes dictated by code.
- Package dependencies
  - Relationships between packages dictated by class relationships.
- Binary dependencies
  - Relationships between deployable units dictated by package relationships.
- To create extensible, maintainable, and testable software, we must manage all types

Agenda

- Introduction
- → Class dependencies
- Package dependencies
- Binary dependencies

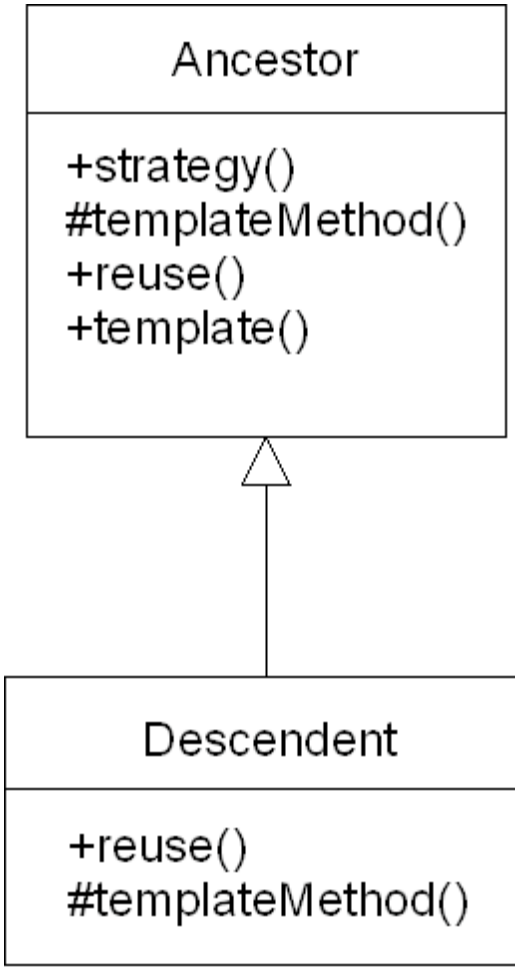
Class Dependencies

- Inheritance
  - Descendent is dependent on ancestor
- Association
  - Bi-directional and uni-directional
  - Abstract vs. concrete
  - Run-time (object) vs. compile-time (class)

Inheritance

- Can be broken using composition
- Inheritance is a static relationship
  - You cannot change your ancestor
- Composition is a dynamic relationship
  - You can change who you associate with
- Inheritance isn't bad, but inheritance for reuse is usually bad.
- Consider refactoring to composition.

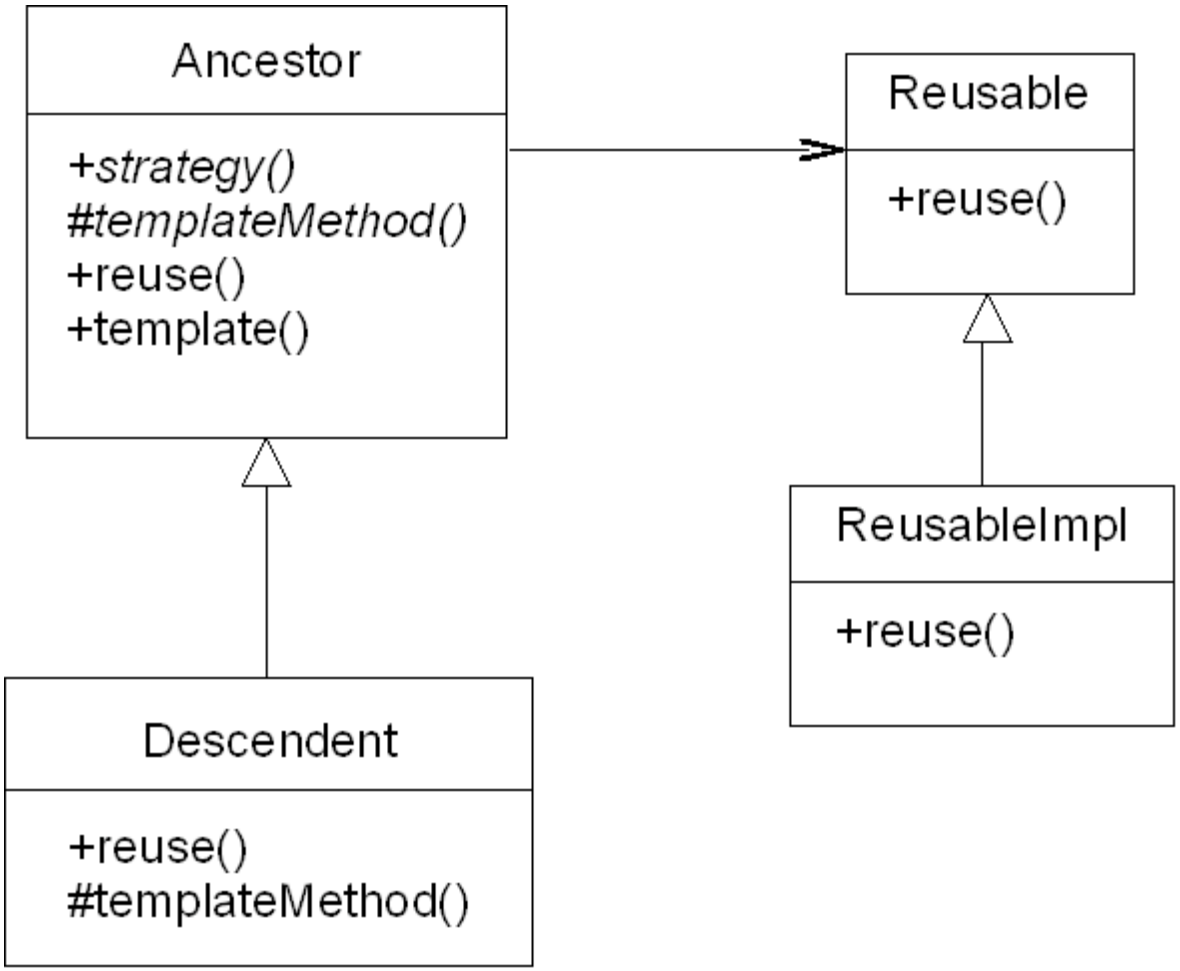
Inheritance Example




```
abstract class Ancestor {
    //Use inheritence for polymorphism.
    public abstract void strategy();
    //Using inheritance for reuse.
    public final void reuse() { System.out.println("Ancestor reuse"); }
    //Use inheritance for reuse & polymorphism.
    protected abstract void templateMethod();
    public void template() {
        System.out.println("Ancestor template");
        this.templateMethod();
    }
}
```

```
class Descendent extends Ancestor {
    protected void templateMethod() {
        System.out.println("Descendent templateMethod");
    }
    public void strategy() { System.out.println("Descendent strategy"); }
}
```

Composition



Composition Code

```
abstract class Ancestor {
    private Reusable reusable;
    public Ancestor(Reusable reusable) {
        this.reusable = reusable;
    }
    //Use inheritence for polymorphism.
    public abstract void strategy();
    //Using inheritance for reuse.
    public final void reuse() { this.reusable.reuse(); }
    //Use inheritance for reuse & polymorphism.
    protected abstract void templateMethod();
    public void template() {
        System.out.println("Ancestor template");
        this.templateMethod();
    }
}
```

```
interface Reusable {
        public void reuse();
}

class ReusableImpl implements Reusable {
    public void reuse() {
        System.out.println("Ancestor reuse");
    }
}
```

```
class Descendent extends Ancestor {
    public Descendent(Reusable reusable) { super(reusable); }
    protected void templateMethod() { System.out.println("Descendent templateMethod"); }
    public void strategy() { System.out.println("Descendent strategy"); }
}
```

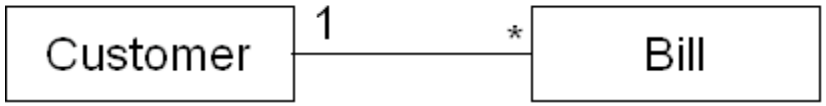
Association

- Favor abstract coupling over concrete coupling
  - Increases testability
- Bi-directional associations can be broken using abstractions
- Remain cognizant of heavy run-time dependencies.
  - They can inhibit reuse due to “configure to use” overload.

Bi-directional Association

- Bi-directional relationship between Customer and Bill.
- Customer has a list of Bill instances. Clients can get total of all Bill charges.
- Bill has a Customer to get the discount amount for that Bill, which is derived based on number of Bills for that Customer.

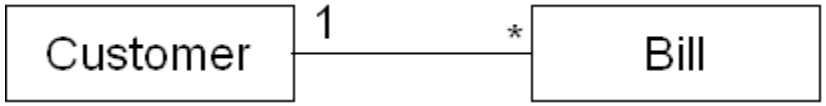
Bi-directional Association




```
import java.util.*;
import java.math.BigDecimal;
public class Customer {
    private List bills;
    public BigDecimal getDiscountAmount() {
        ...code...
    }
    public List getBills() {  return this.bills;  }
    public void createBill(BigDecimal chargeAmount) {
        ...code...
    }
}
```

```
import java.math.BigDecimal;
public class Bill {
    private BigDecimal chargeAmount;
    private Customer customer;
    public Bill(Customer customer,
                BigDecimal chargeAmount) {
        this.customer = customer;
        this.chargeAmount = chargeAmount;
    }
    public BigDecimal pay() {
        BigDecimal discount = new BigDecimal(1).
            subtract(this.customer.getDiscountAmount()).
            setScale(2, BigDecimal.ROUND_HALF_UP);
        //make the payment...
        return paidAmount;
    }
}
```

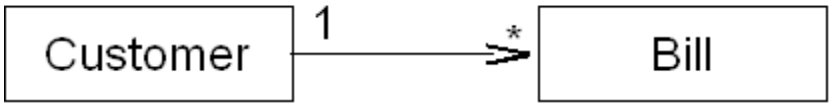
Bi-directional Assocation




```
public void testPayment() {
    Customer customer = new Customer();
    customer.createBill(new BigDecimal(500));
    Iterator bills = customer.getBills().iterator();
    while (bills.hasNext()) {
        Bill bill = (Bill) bills.next();
        BigDecimal paidAmount = bill.pay();
        assertEquals("Paid amount not correct.", new BigDecimal(485).setScale(2), paidAmount);
    }
}
```

- Test creates Customer and Bill.
- Loops through all Bill instances and makes payment.

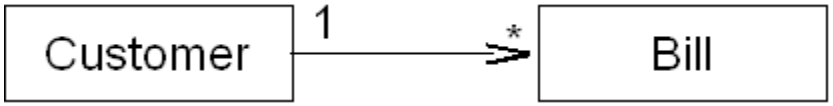
Uni-directional Association




```
import java.util.*;
import java.math.BigDecimal;
public class Customer {
    private List bills;
    public BigDecimal getDiscountAmount() {
        ...code...
    }
    public List getBills() {  return this.bills;  }
    public void createBill(BigDecimal chargeAmount) {
        ...code...
    }
}
```

```
import java.math.BigDecimal;
public class Bill {
    private BigDecimal chargeAmount;
    public Bill(BigDecimal chargeAmount) {
        this.chargeAmount = chargeAmount;
    }
    public BigDecimal getChargeAmount() {
        return this.chargeAmount;
    }
    public BigDecimal pay(BigDecimal discountAmount) {
        //make the payment...
        return paidAmount;
    }
}
```

Uni-directional Association



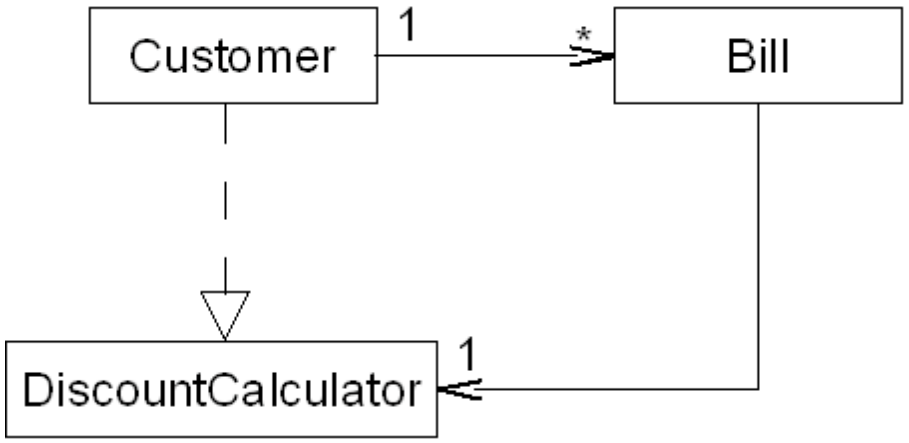

```
public void testPayment() {
    Customer customer = new Customer();
    customer.createBill(new BigDecimal(500));
    Iterator bills = customer.getBills().iterator();
    BigDecimal discount = new BigDecimal(1).subtract(customer.getDiscountAmount()).
        setScale(2, BigDecimal.ROUND_HALF_UP);
    while (bills.hasNext()) {
        Bill bill = (Bill) bills.next();
        BigDecimal paidAmount = bill.pay(discount);
        assertEquals("Paid amount not correct.", new BigDecimal(485).setScale(2), paidAmount);
    }
```

- Bi-directional association has been eliminated.
- The test must have more knowledge (calculate the discount).

Compare/Constrast

- Bi-directional – Client knows less about internal behavior of Customer and Bill (doesn't need to calculate the discount).
- Uni-directional – Bill is perceivably more reusable, but the real advantage is maintainability and testability.
- Very little extensibility with either approach.

Abstract Association



- All uni-directional run-time relationships.
- Advantage of client not needing to calculate discount.
- Advantage of easier maintainability and extensibility.

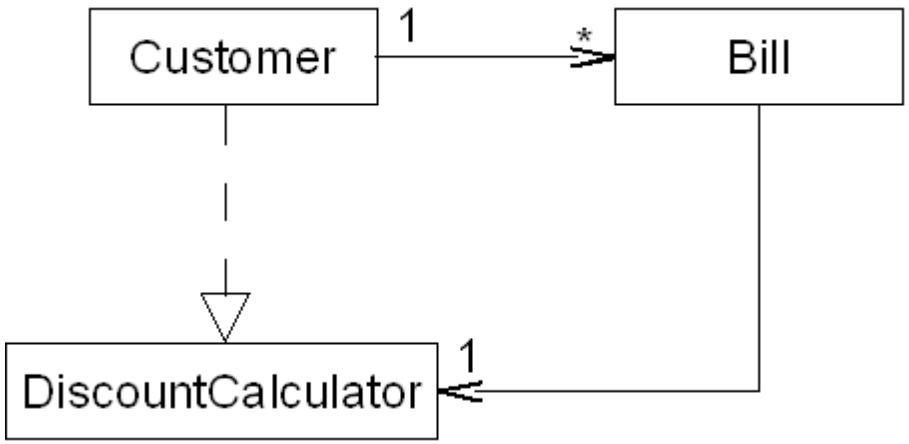
Abstract Association

```
import java.util.*;
import java.math.BigDecimal;
public class Customer implements DiscountCalculator {
    private List bills;
    public BigDecimal getDiscountAmount() {
        ... code ...
    }
    public List getBills() {    return this.bills;    }
    public void createBill(BigDecimal chargeAmount) {
        ... code ...
    }
}
```

```
import java.math.BigDecimal;

public interface DiscountCalculator {
            public BigDecimal getDiscountAmount();
}
```

```
import java.math.BigDecimal;
public class Bill {
    private BigDecimal chargeAmount;
    private DiscountCalculator discounter;
    public Bill(DiscountCalculator discounter,
                BigDecimal chargeAmount) {
        this.discounter = discounter;
        this.chargeAmount = chargeAmount;
    }
    public BigDecimal pay() {
        BigDecimal discount = new BigDecimal(1).
            subtract(this.discounter.getDiscountAmount()).
            setScale(2, BigDecimal.ROUND_HALF_UP);
        BigDecimal paidAmount = this.chargeAmount.
             multiply(discount).setScale(2);
        //make the payment...
        return paidAmount;
    }
}
```

Abstract Association

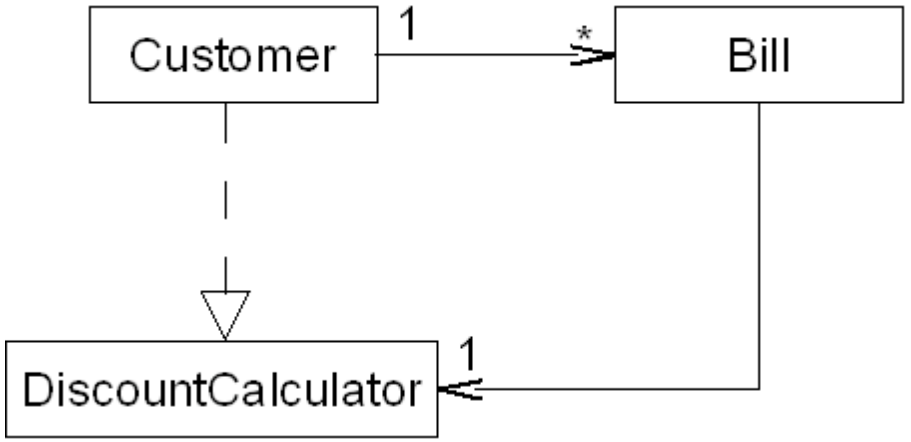


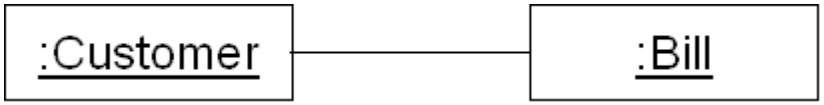

```
public void testPayment() {
    Customer customer = new Customer();
    customer.createBill(new BigDecimal(500));
    Iterator bills = customer.getBills().iterator();
    while (bills.hasNext()) {
        Bill bill = (Bill) bills.next();
        BigDecimal paidAmount = bill.pay();
        assertEquals("Paid amount not correct.",
            new BigDecimal(485).setScale(2), paidAmount);
    }
}
```

- Ability to test Bill independent of Customer.

```
public void testPaymentWithoutCustomer() {
    Bill bill = new Bill(new DiscountCalculator() {
        public BigDecimal getDiscountAmount() {
            return new BigDecimal(0.1); }
        }, new BigDecimal(500));
    BigDecimal paidAmount = bill.pay();
    assertEquals("Paid amount not correct.", new BigDecimal(450).
        setScale(2), paidAmount);
}
```

Run-time vs. Compile-time





- In most situations, we still have a bi-directional run-time relationship between Customer and Bill instances.
- Is this bad?

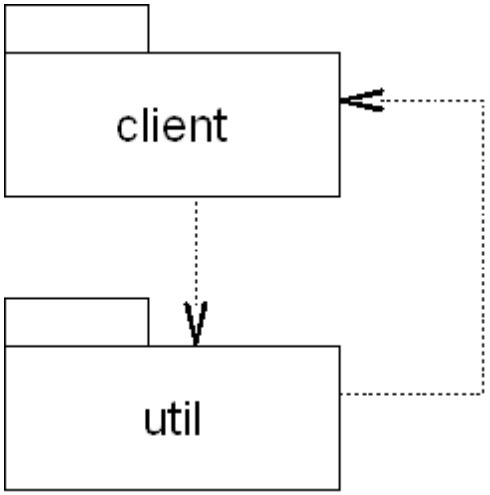
Agenda

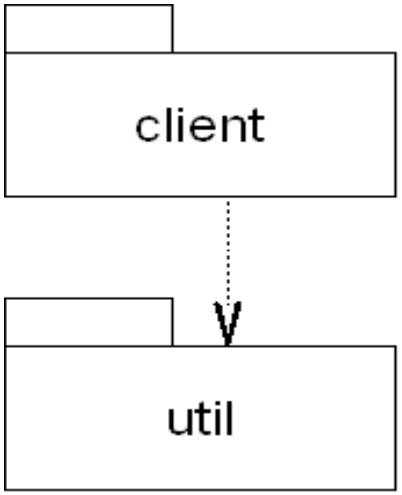
- Introduction
- Class dependencies
- → Package dependencies
- Binary dependencies

Package Dependencies

- Modeled using UML Package diagrams
- Specified directly in Java using import
- Direct and Indirect Dependencies
- Cyclic versus Acyclic Dependencies
- JDepend
  - Include output reports in your build
  - Create test cases to verify dependency structure

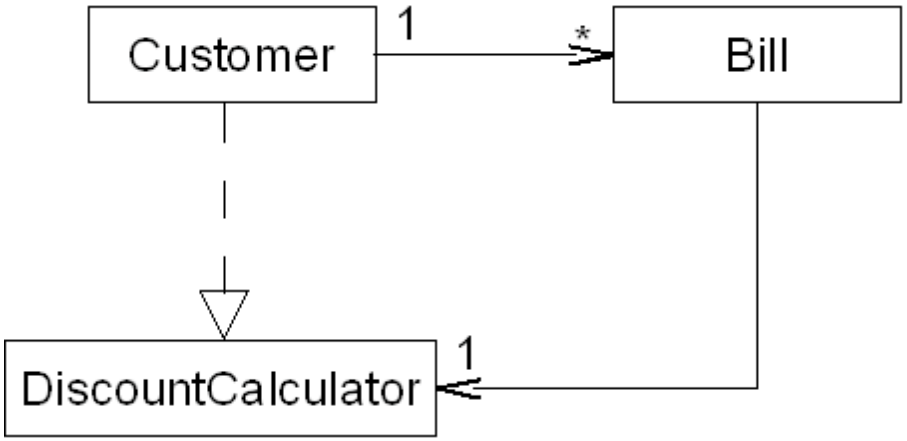
Cyclic vs. Acyclic



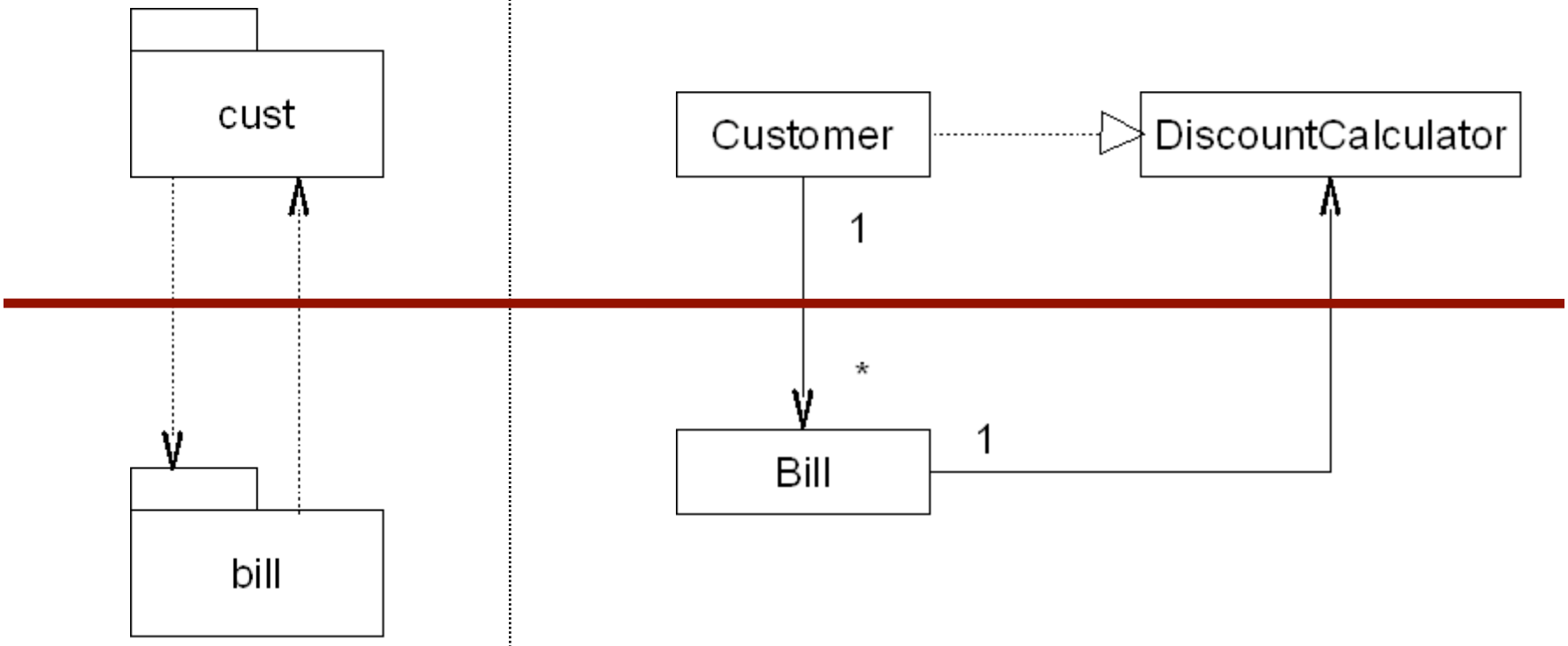


Packaging

- What would be a good package structure?



Initial Packaging



Interface should be “close” to the class that “uses” it.

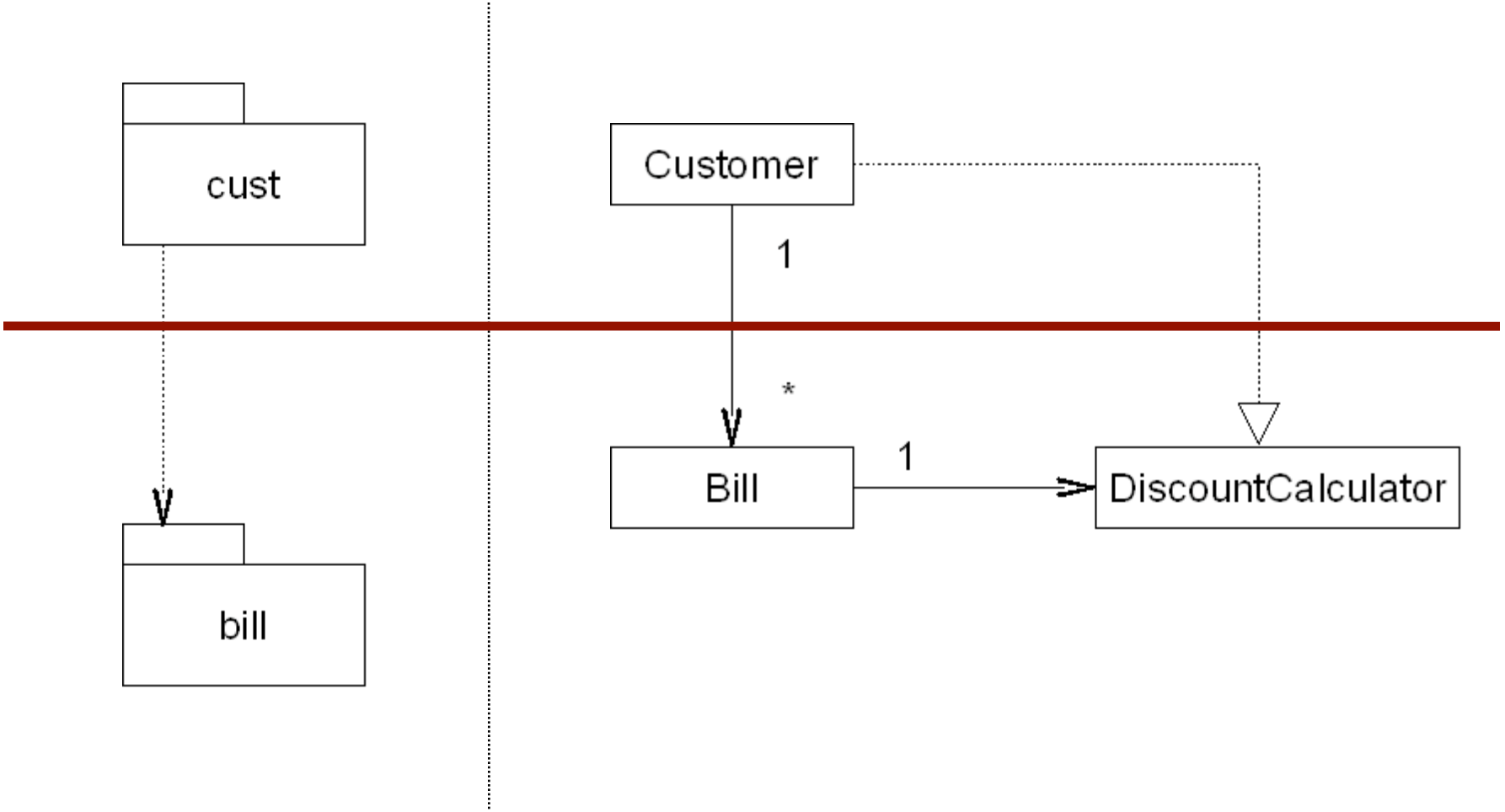
Verify Dependencies

```
public class DependencyConstraintTest extends TestCase {
   private JDepend jDepend;

   public static void main(String[] args) {
      junit.textui.TestRunner.run(DependencyConstraintTest.class);
   }
   public DependencyConstraintTest(String name) {    super(name);    }
      protected void setUp() throws Exception {
         PackageFilter filter = new PackageFilter();
         filter.addPackage("java.*");
         jDepend = new JDepend(filter);
         jDepend.addDirectory("build/com/kirkk/cust");
         jDepend.addDirectory("build/com/kirkk/bill");
      }
      public void testPackageDependencies() {
         DependencyConstraint constraint = new DependencyConstraint();
         JavaPackage cust = constraint.addPackage("com.kirkk.cust");
         JavaPackage bill = constraint.addPackage("com.kirkk.bill");
         cust.dependsUpon(bill);
         bill.dependsUpon(cust);
         jDepend.analyze();
         assertEquals("Dependency Mismatch", true, jDepend.dependencyMatch(constraint));
      }
}
```

Setup directories and packages to exclude.

Add the constraints
NOTE: Must add all dependencies or test will fail.

Acyclic Package Structure



Verify Dependencies

```
public class DependencyConstraintTest extends TestCase {
   private JDepend jDepend;

   public static void main(String[] args) {
      junit.textui.TestRunner.run(DependencyConstraintTest.class);
   }
   public DependencyConstraintTest(String name) {    super(name);    }
      protected void setUp() throws Exception {
         PackageFilter filter = new PackageFilter();
         filter.addPackage("java.*");
         jDepend = new JDepend(filter);
         jDepend.addDirectory("build/com/kirkk/cust");
         jDepend.addDirectory("build/com/kirkk/bill");
      }
      public void testPackageDependencies() {
         DependencyConstraint constraint = new DependencyConstraint();
         JavaPackage cust = constraint.addPackage("com.kirkk.cust");
         JavaPackage bill = constraint.addPackage("com.kirkk.bill");
         cust.dependsUpon(bill);
         bill.dependsUpon(cust);        //remove this line
         jDepend.analyze();
         assertEquals("Dependency Mismatch", true, jDepend.dependencyMatch(constraint));
      }
}
```

Testing Cycles

```
public void testAllDependencies() {
    jDepend.analyze();
    java.util.Iterator i = jDepend.getPackages().iterator();
    ArrayList cyclePackages = new ArrayList();
    while (i.hasNext()) {
        JavaPackage p = (JavaPackage) i.next();
        if (p.containsCycle()) {
            cyclePackages.add(p);
        }
    }
    assertEquals("Cycles exist: " + getCycles(cyclePackages), 0, cyclePackages.size());
}
```

- Test will fail if any cyclic dependencies exist.

JDepend Build Report

```
<target name="jdepend" depends="compile">
   <jdepend format="xml" outputfile="${buildstats}/jdepend.xml">
      <classespath>
         <pathelement location="${build}" />
      </classespath>
      <classpath location=""/>
   </jdepend>

   <style in="${buildstats}/jdepend.xml"
            out="${buildstats}/jdepend.html"
            style="${lib}/jdepend.xsl">
   </style>
</target>
```

- Generates html document with detailed Jdepend analysis.
- Cyclic dependencies do not cause build failure.

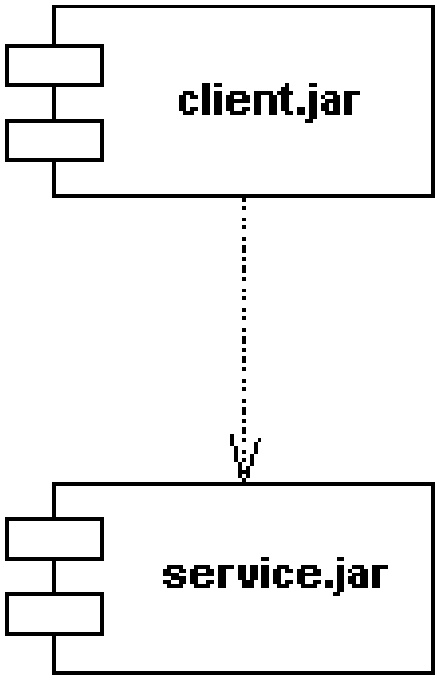
Agenda

- Introduction
- Class dependencies
- Package dependencies
- → Binary dependencies

Binary Dependencies

- Binary units of deployment are the unit of reuse
- Direct and Indirect Dependencies
- Cyclic versus Acyclic Dependencies
- Inverting Dependencies
- Specified by classpath
- JarAnalyzer
  - Include output reports and component diagram in your build

Direct Dependency

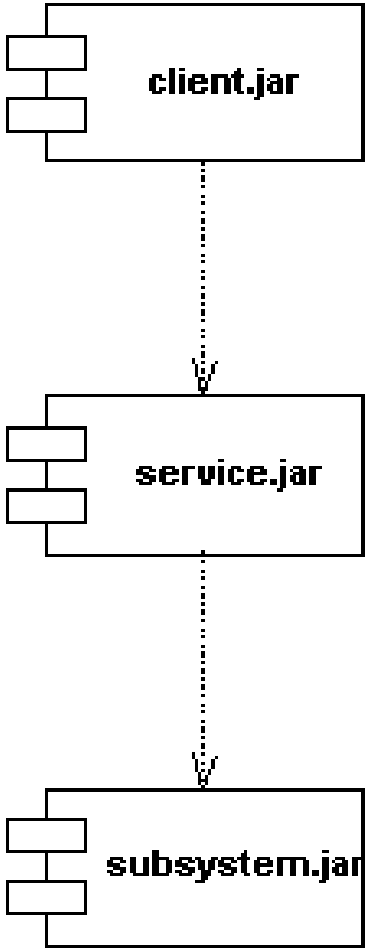



```
package client;
import service.Service;
public class Client {

}
```

```
package service;
public class Service {

}
```

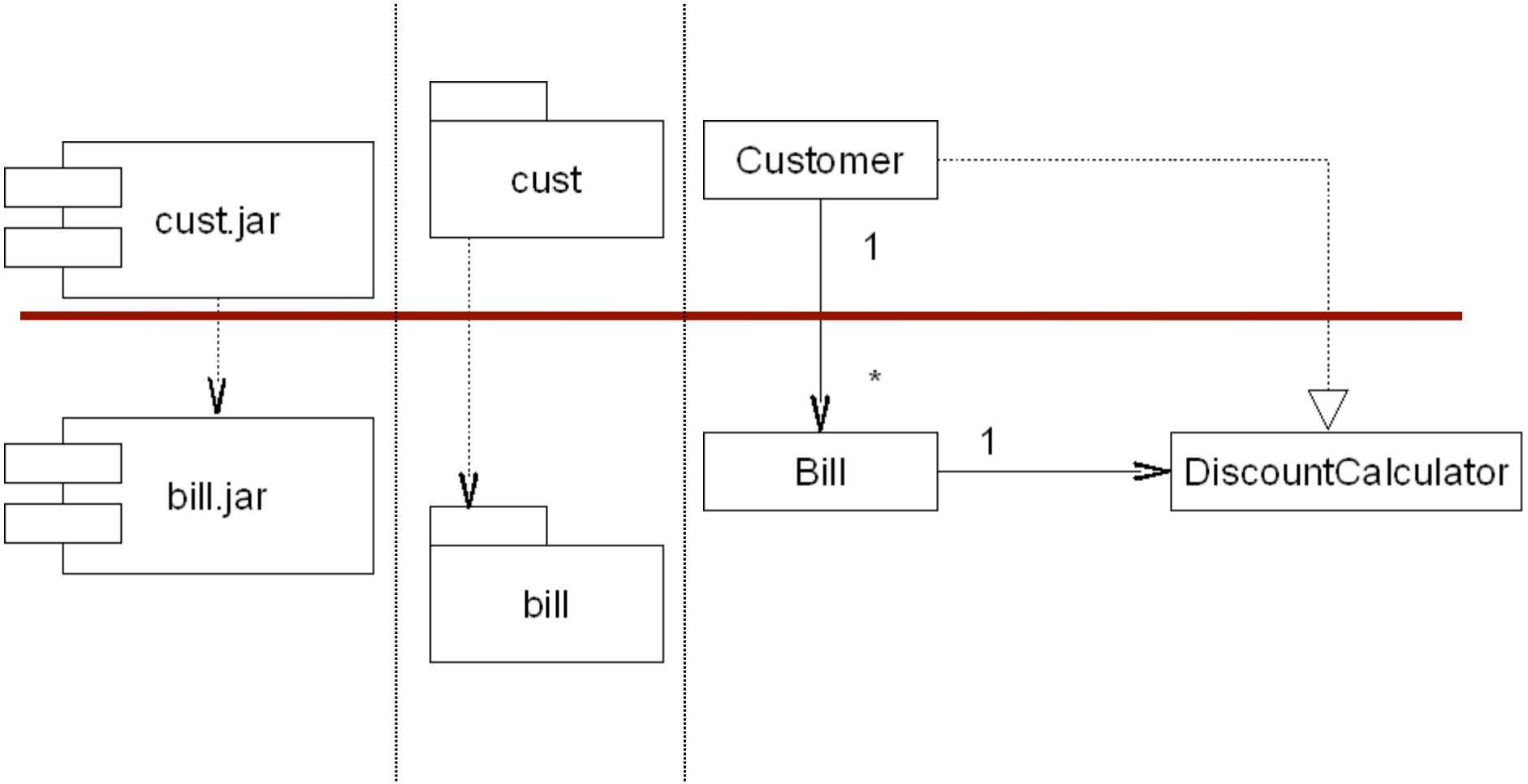
The client component cannot be deployed without the service component.

Indirect Dependency

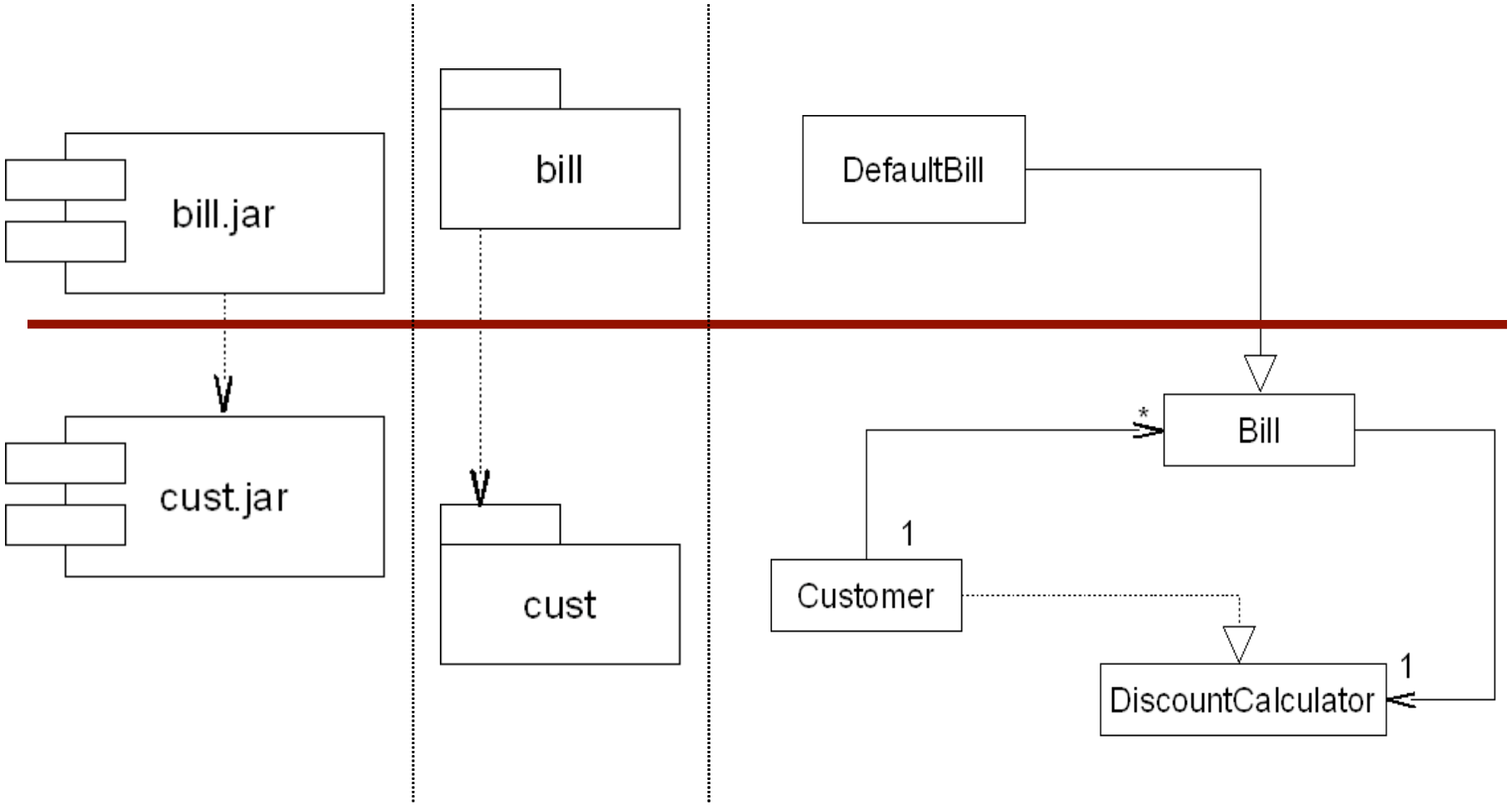


The client component cannot be deployed without the service or subsystem component.

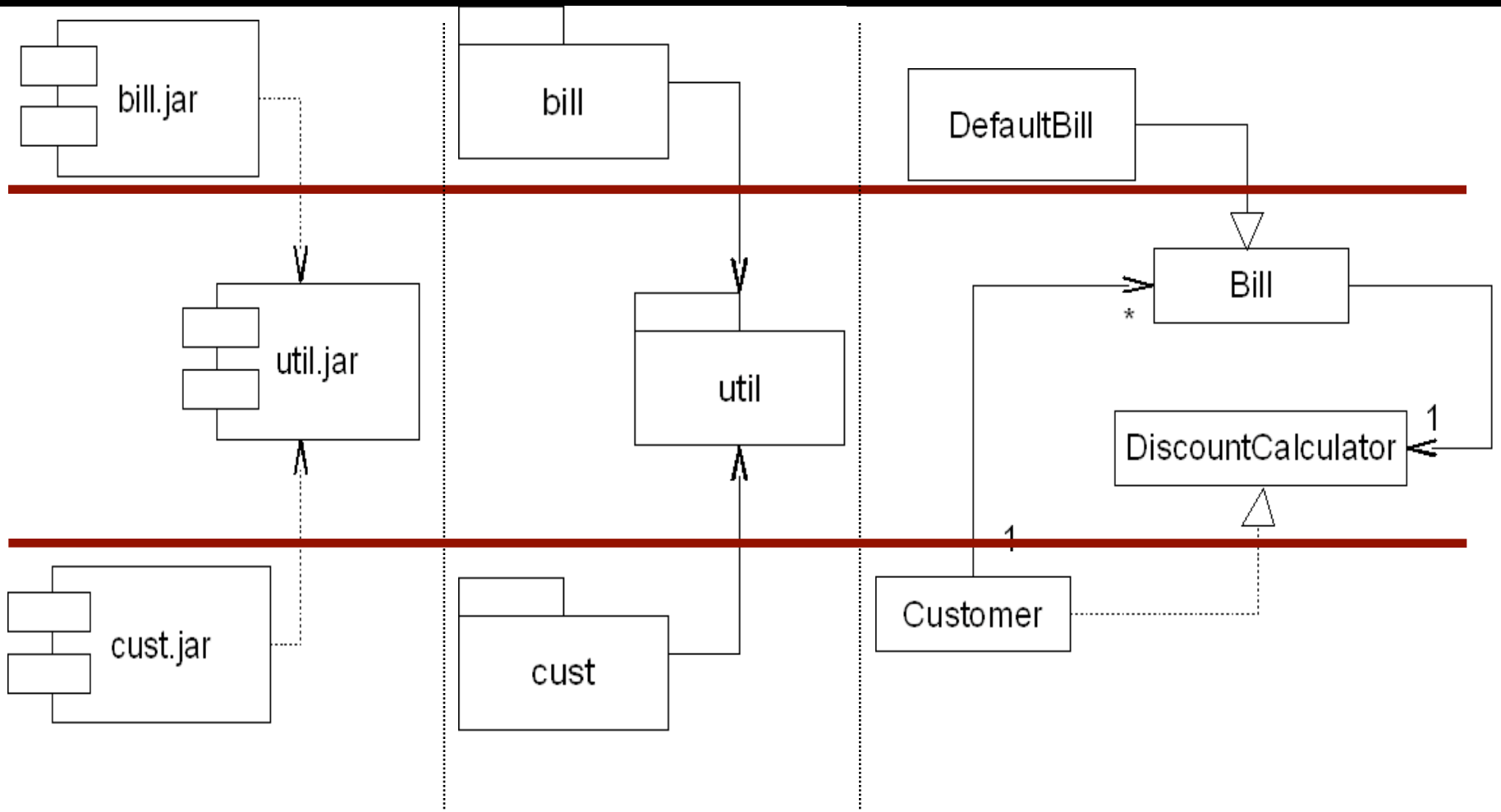
Direct


cust.jar
cust
Customer
1
*
bill.jar
bill
Bill
1
DiscountCalculator

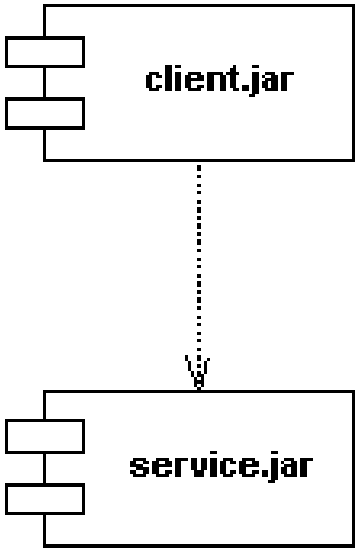
Inverted


bill.jar
bill
DefaultBill
cust.jar
cust
*
Bill
1
Customer
1
DiscountCalculator

Eliminated


bill.jar
bill
DefaultBill
Bill
*
util.jar
util
DiscountCalculator
1
1
cust.jar
cust
Customer

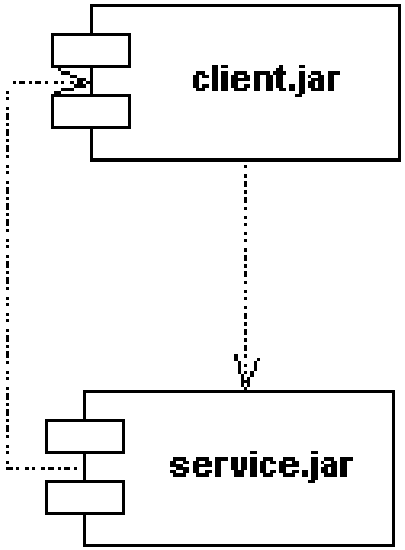
Cyclic and Acyclic Dependencies




```
package client;
import service.Service;
public class Client {

}
```

```
package service;
public class Service {

}
```

Uni-Directional Component Relationship




```
package client;
import service.Service;
public class Client {

}
```

```
package service;
public class Service {

}
```

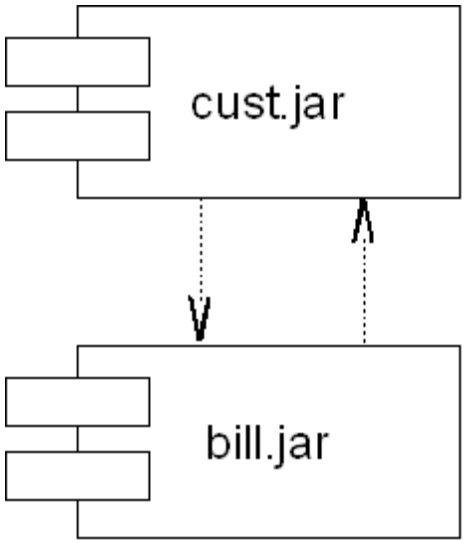
```
package service;
import client.Client;
public class Impl {

}
```

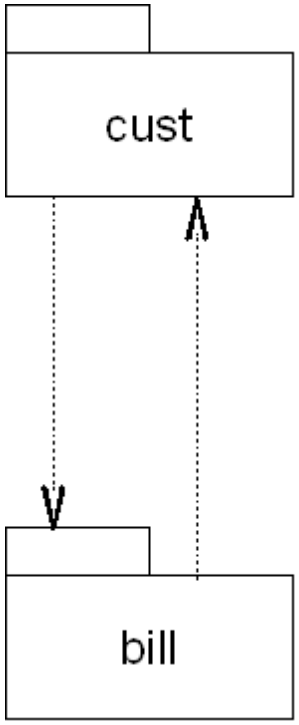
Bi-Directional Component Relationship

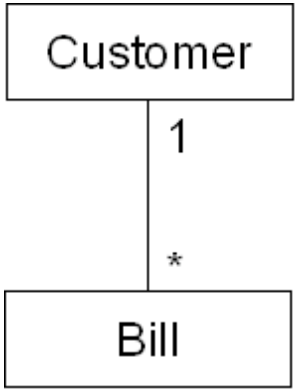
Eliminating Cycles

- Escalation
  - Move cause of dependency to higher level component.
- Demotion
  - Move dependent class to lower level component.
- Callback
  - "Observer" like behavior.

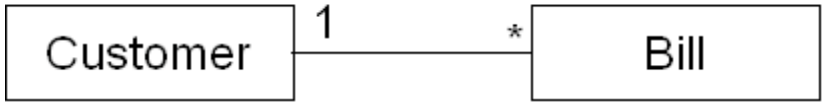
Original Structure







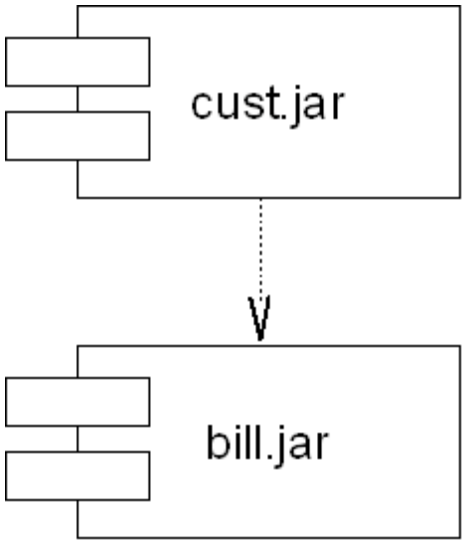
Bi-directional Association

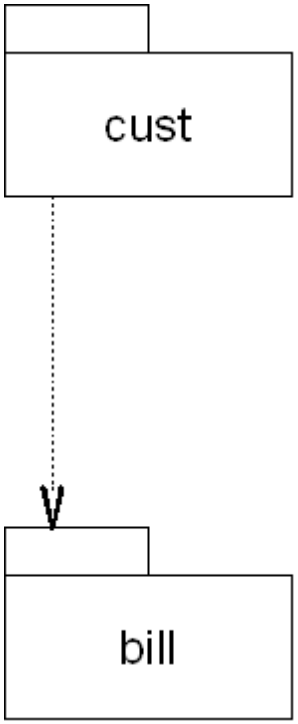


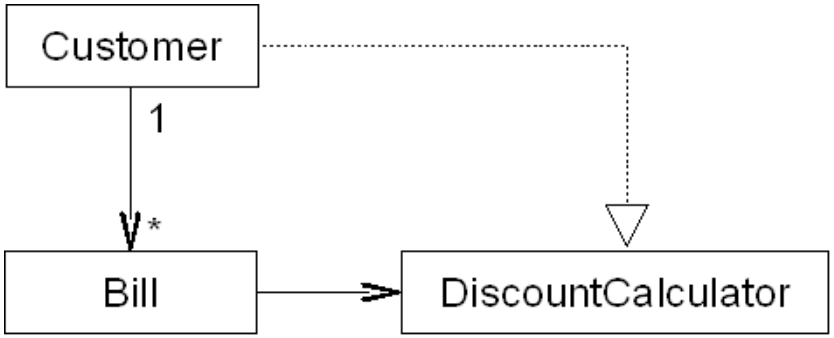

```
import java.util.*;
import java.math.BigDecimal;
import com.kirkk.bill.*;
public class Customer {
    private List bills;
    public BigDecimal getDiscountAmount() {
        ...code...
    }
    public List getBills() {  return this.bills;  }
    public void createBill(BigDecimal chargeAmount) {
        ...code...
    }
}
```

```
import java.math.BigDecimal;
import com.kirkk.cust.*;
public class Bill {
    private BigDecimal chargeAmount;
    private Customer customer;
    public Bill(Customer customer,
                BigDecimal chargeAmount) {
        this.customer = customer;
        this.chargeAmount = chargeAmount;
    }
    public BigDecimal pay() {
        BigDecimal discount = new BigDecimal(1).
            subtract(this.customer.getDiscountAmount()).
            setScale(2, BigDecimal.ROUND_HALF_UP);
        //make the payment...
        return paidAmount;
    }
}
```

Callback


cust.jar
bill.jar


cust
bill


Customer
1
*
Bill
DiscountCalculator

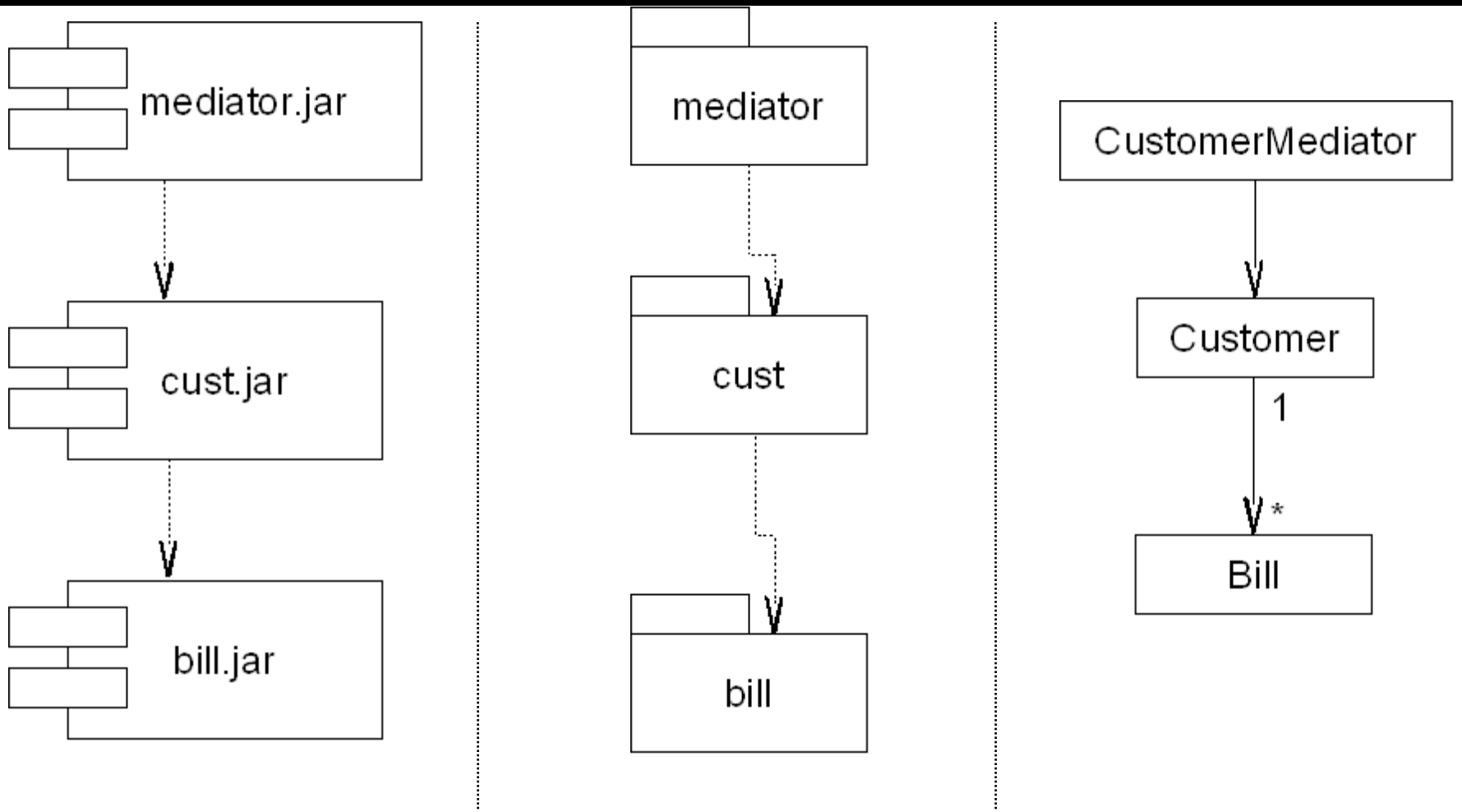
Abstract Association

```
import java.util.*;
import java.math.BigDecimal;
import com.kirkk.bill.*;
public class Customer implements DiscountCalculator {
    private List bills;
    public BigDecimal getDiscountAmount() {
        ... code ...
    }
    public List getBills() {    return this.bills;    }
    public void createBill(BigDecimal chargeAmount) {
        ... code ...
    }
}
```

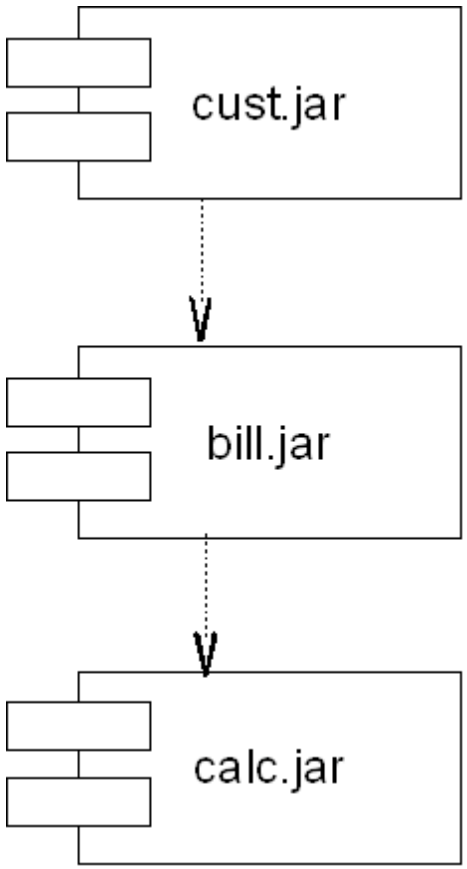
```
import java.math.BigDecimal;

public interface DiscountCalculator {
            public BigDecimal getDiscountAmount();
}
```

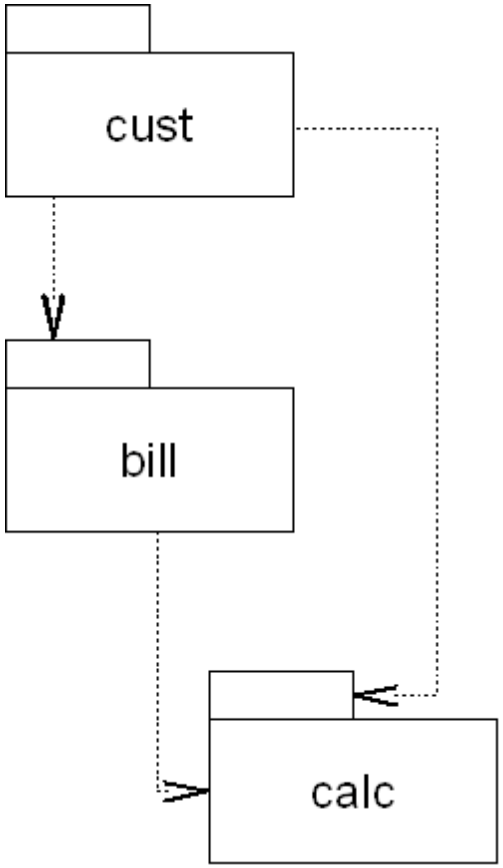
```
import java.math.BigDecimal;
public class Bill {
    private BigDecimal chargeAmount;
    private DiscountCalculator discounter;
    public Bill(DiscountCalculator discounter,
                BigDecimal chargeAmount) {
        this.discounter = discounter;
        this.chargeAmount = chargeAmount;
    }
    public BigDecimal pay() {
        BigDecimal discount = new BigDecimal(1).
            subtract(this.discounter.getDiscountAmount()).
            setScale(2, BigDecimal.ROUND_HALF_UP);
        BigDecimal paidAmount = this.chargeAmount.
             multiply(discount).setScale(2);
        //make the payment...
        return paidAmount;
    }
}
```

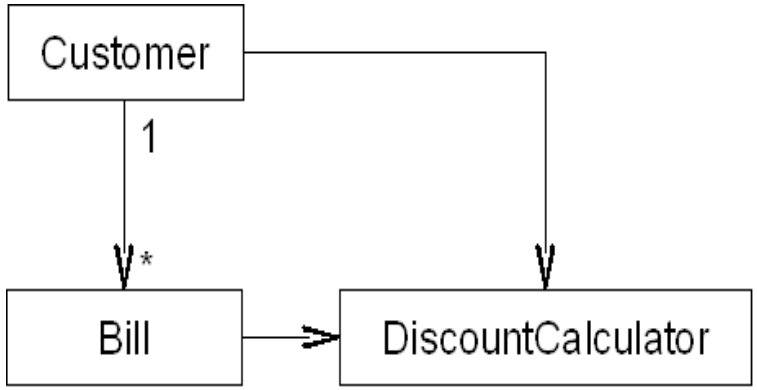
Escalation


mediator.jar
cust.jar
bill.jar
mediator
cust
bill
CustomerMediator
Customer
1
*
Bill

Demotion


cust.jar
bill.jar
calc.jar


cust
bill
calc


Customer
1
*
Bill
DiscountCalculator

JarAnalyzer

- Open source utility for analyzing the relationships between .jar files.
- Visual output using GraphViz
- .xml output also available with more detailed metrics.

Additional Resources

- www.kirkk.com
  - JarAnalyzer download and general information on software development.
- www.extensiblejava.com
  - Resource devoted exclusively to dependency management.

Please complete your session evaluation forms