From Code To Architecture

Kirk Knoernschild

Chief Technology Strategist

QWAN*tify*, Inc.

www.qwantify.com

kirk@qwantify.com

Agenda

- Attempt the impossible – Define Architecture
- Logical vs. Physical Design
- Component Heuristics

Software Architecture

- An architecture is the set of **significant decisions about the organization of a software system**, the selection of **the structural elements and their interfaces** by which the system is composed, **together with their behavior** as specified in the collaborations among those elements, the **composition of these structural and behavioral elements into progressively larger subsystems**, and the architectural style that guides this organization---these elements and their interfaces, their collaborations, and their composition (Kruchten: The Rational Unified Process. Also cited in Booch, Rumbaugh, and Jacobson: The Unified Modeling Language User Guide, Addison-Wesley, 1999).

Software Architecture

- In most successful software projects, **the expert developers working on that project have a shared understanding of the system design**. This shared understanding is called 'architecture.' This understanding includes how **the system is divided into components** and **how the components interact through interfaces**. These components are usually composed of smaller components, but **the architecture only includes the components and interfaces that are understood by all the developers**...Architecture is about the important stuff. Whatever that is. (Fowler, Martin. IEEE Software, 2003. "Who Needs and Architect.") Quoting Ralph Johnson from the XP mailing list.

Software Architecture

- Structure
- Subsystems and components
- Interfaces
- Your code defines the structure, is pulled together to create subsystems and components, and is decoupled using interfaces.
- How is this so?

Heuristics

- General "rules of thumb" offering guidance in most usual situations.
- Not *always* appropriate.
  - Is the "usual" case.
- Foundation of many common patterns.
- Techniques helping us tailor pattern to context.

Goals of Development

- Maximize reuse
  - Faster development
- Ease maintenance
  - Less error prone changes
  - Faster changes
- Coupling and Cohesion must always be managed.

Logical Design

- Relationship between classes.
- Two types of relationships
  - Dependency
  - Inheritance
- Emphasis on maintenance and extensibility.
- Relevant on all size systems.

Physical Design

- Structure of files (.jar) and directories (packages).
- Dependencies exist between these structures.
- Emphasis on reuse, build, and deploy.
- Relevant mainly to large systems.
  - Modularity to resolve complexity

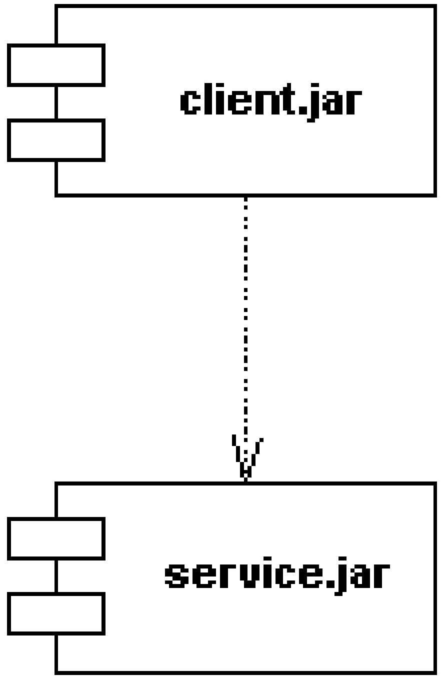
POJO Components

- Binary unit of deployment
  - .jar file
- Relationships build upon each other
  - Class relationships enable
  - Package relationships enforce
  - Component relationships achieve
- Aren't coupled to a container

ComponentRelationships

- "Design component relationships"
- *If changing the contents of a component, C2, may impact the contents of another component, C1, we can say that C1 has a Physical Dependency on C2. [JOUP02]*

Direct Dependency

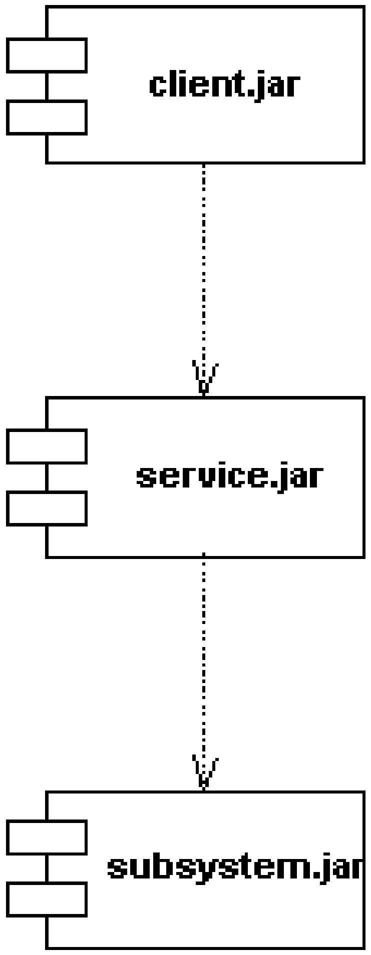


```
package client;
import service.Service;
public class Client {

}
```

```
package service;
public class Service {

}
```

The client component cannot be deployed without the service component.

Indirect Dependency



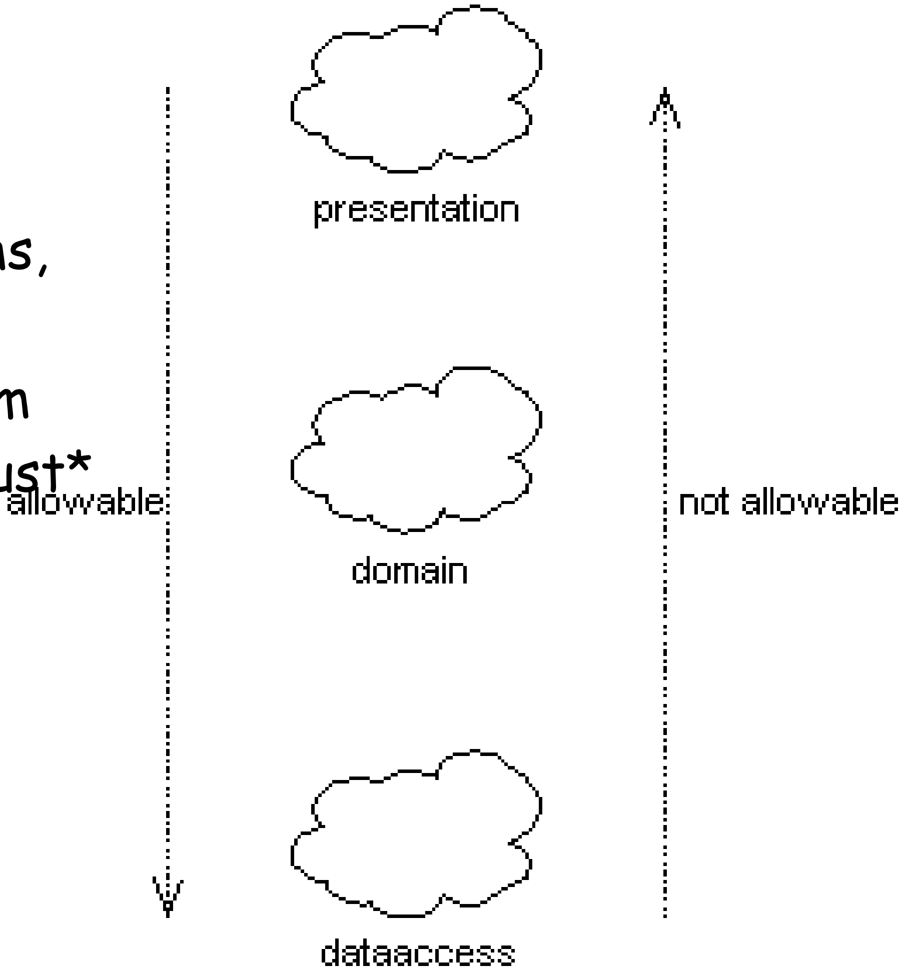
The client component cannot be deployed without the service or subsystem component.

PhysicalLayers

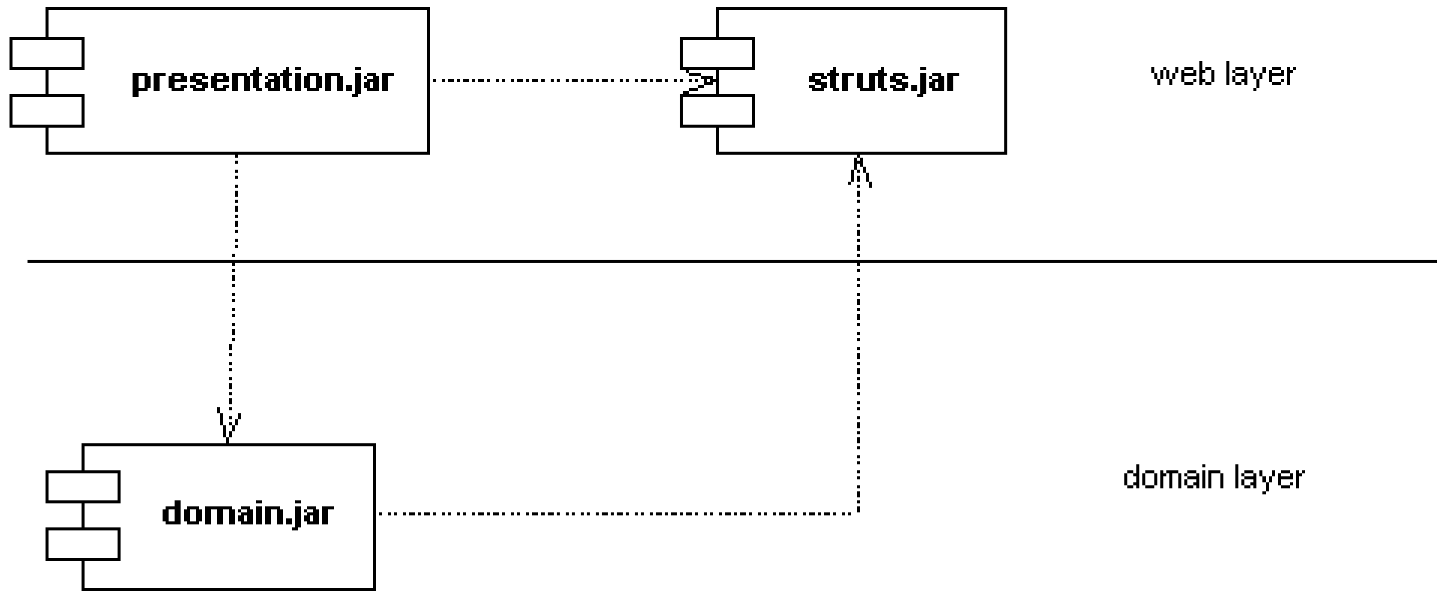
- “Component relationships should not violate the logical layers.”
- Common logical layers
  - Presentation
  - Business Logic
  - Data Access

Logical Layers

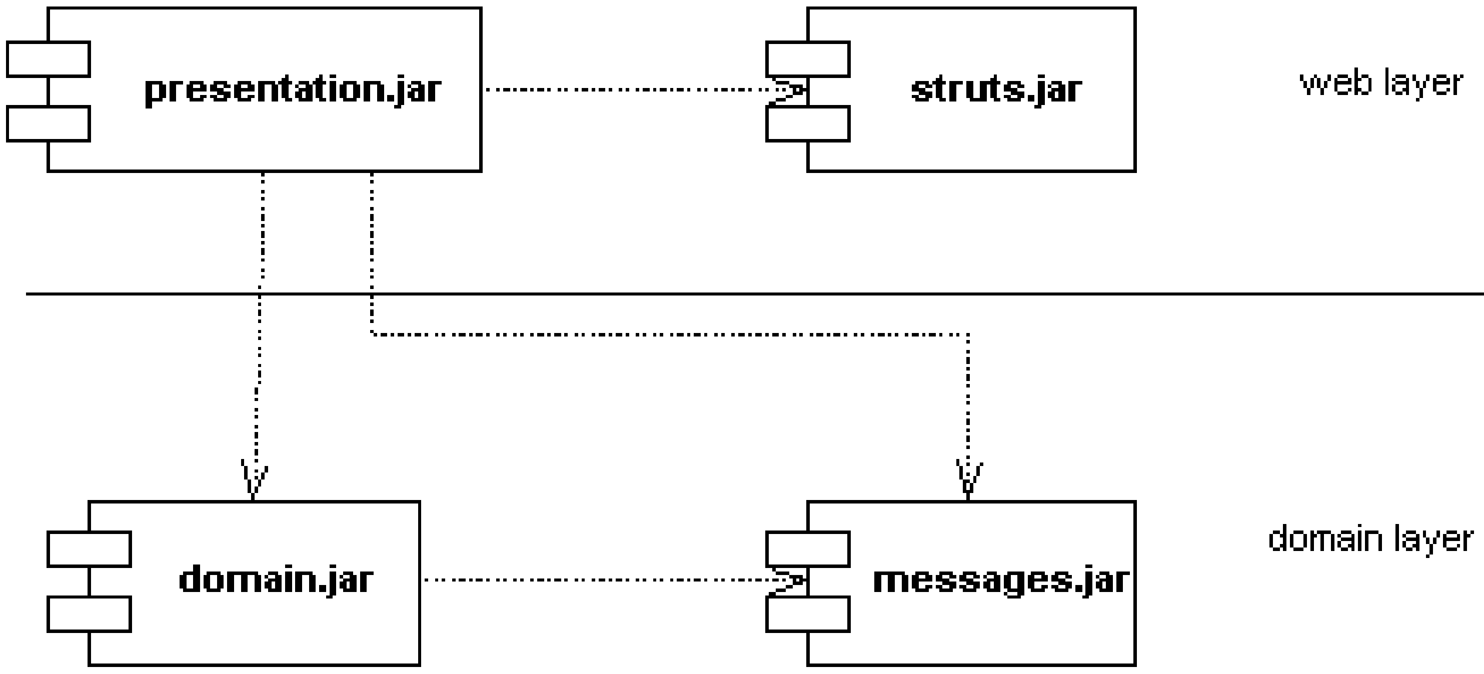
For small systems, all may be in the same .jar file. For larger systems, breaking these apart can increase reusability. And if you do break them apart, the physical relationships *must* be enforced.



Layer Violation



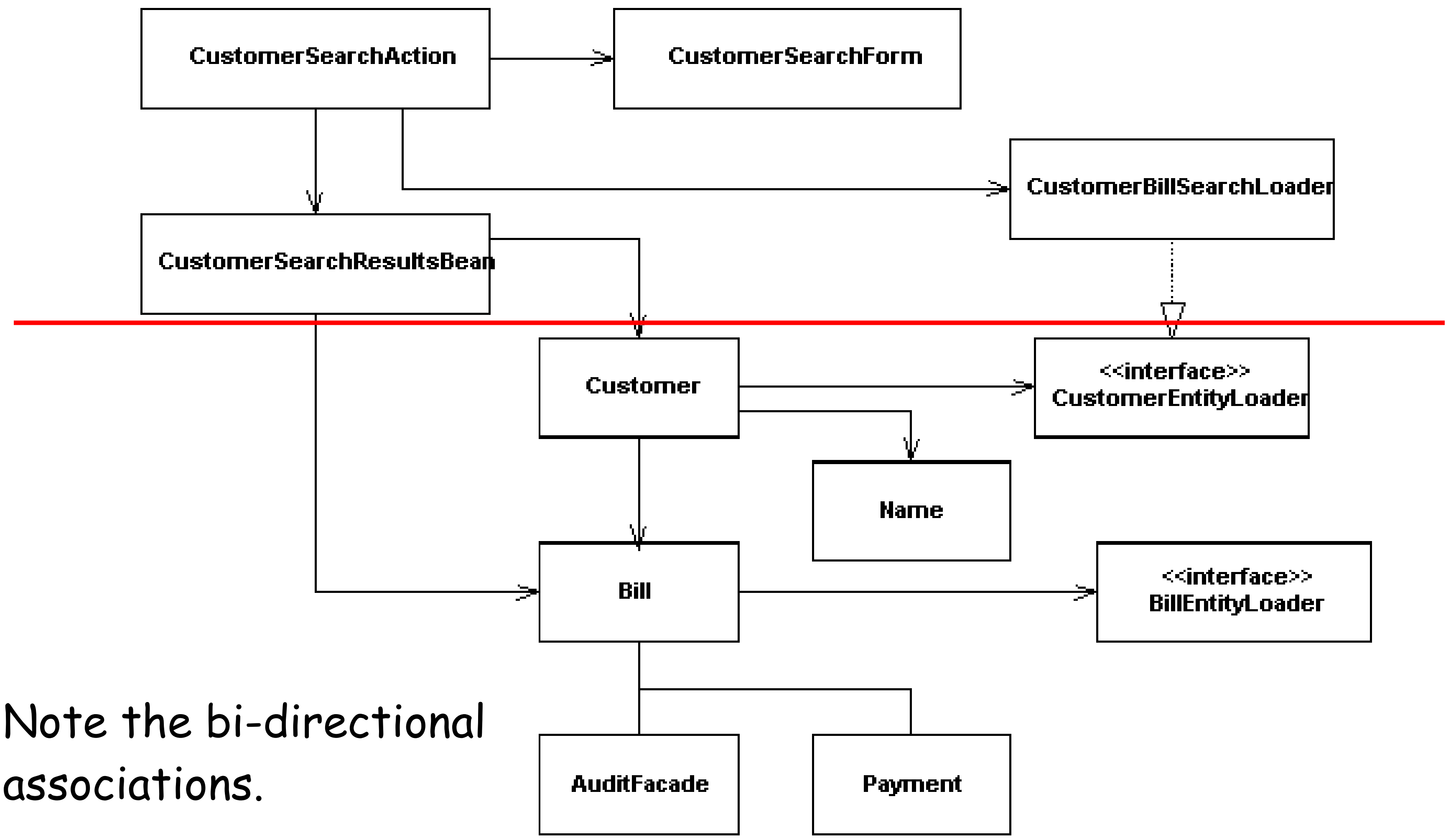
Violation Corrected



BillPay System

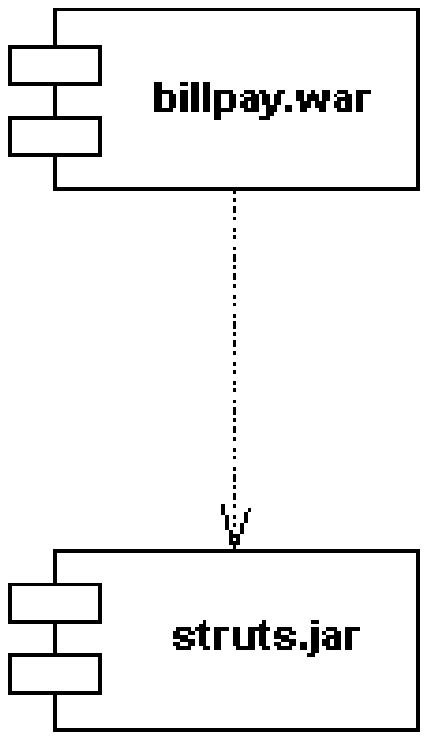
- Design a system to handle payment and auditing of various types of bills. The system must integrate with 3rd party auditing software, and a legacy financials system that must be fed payment information for reconciliation.

Version 1 Class Diagram

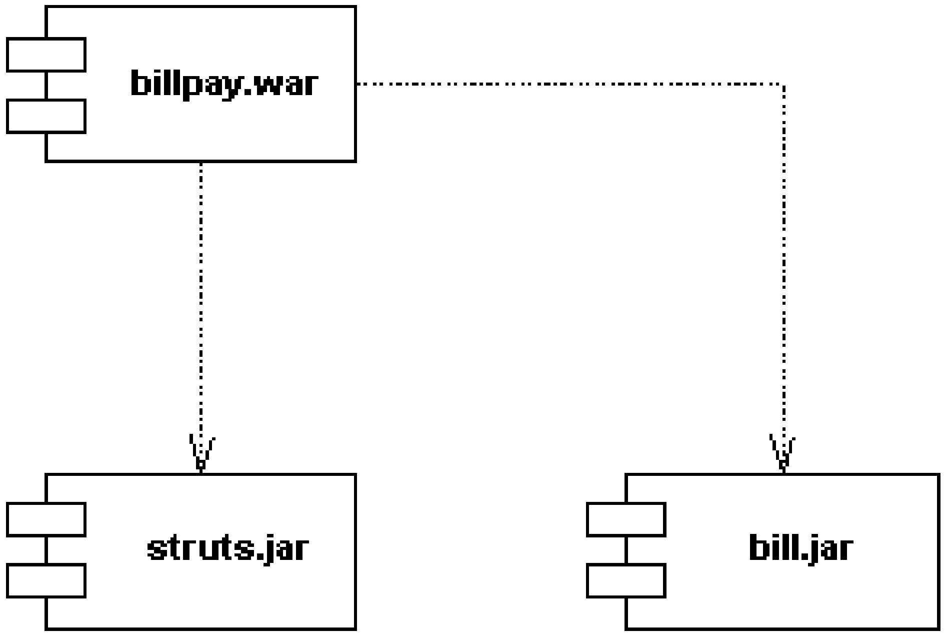


Note the bi-directional associations.

Version 1 Component Diagram



Physical Separation



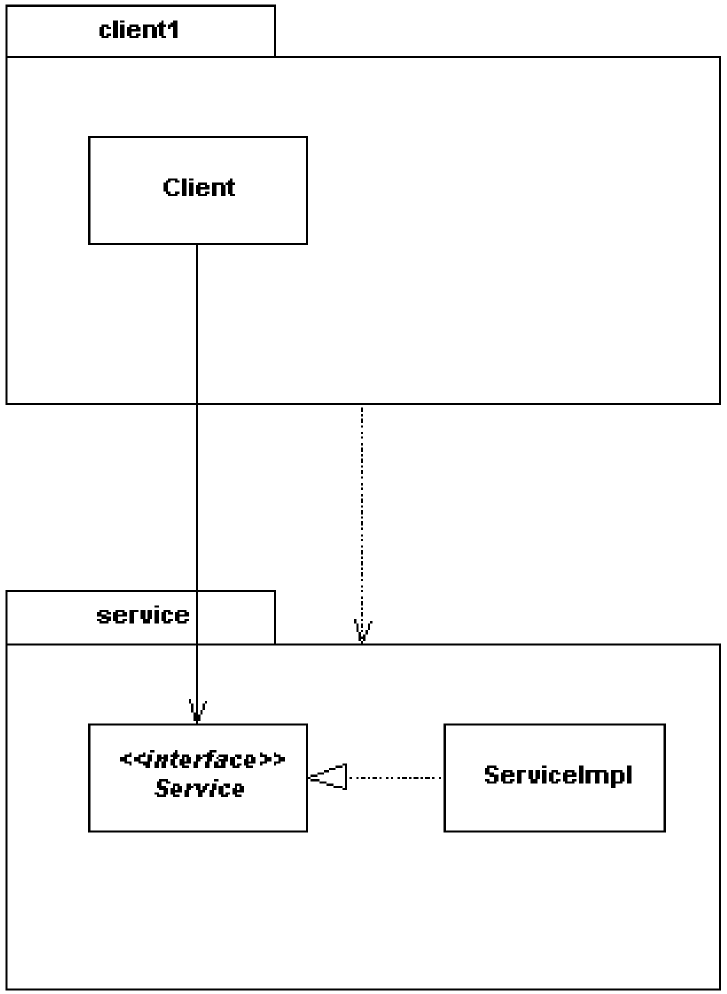
AbstractComponents

- “Depend upon the abstract elements of a component.”
- In other words, depend on abstract classes or interfaces.

www.qwantify.com

Abstract Dependency

client.jar



service.jar

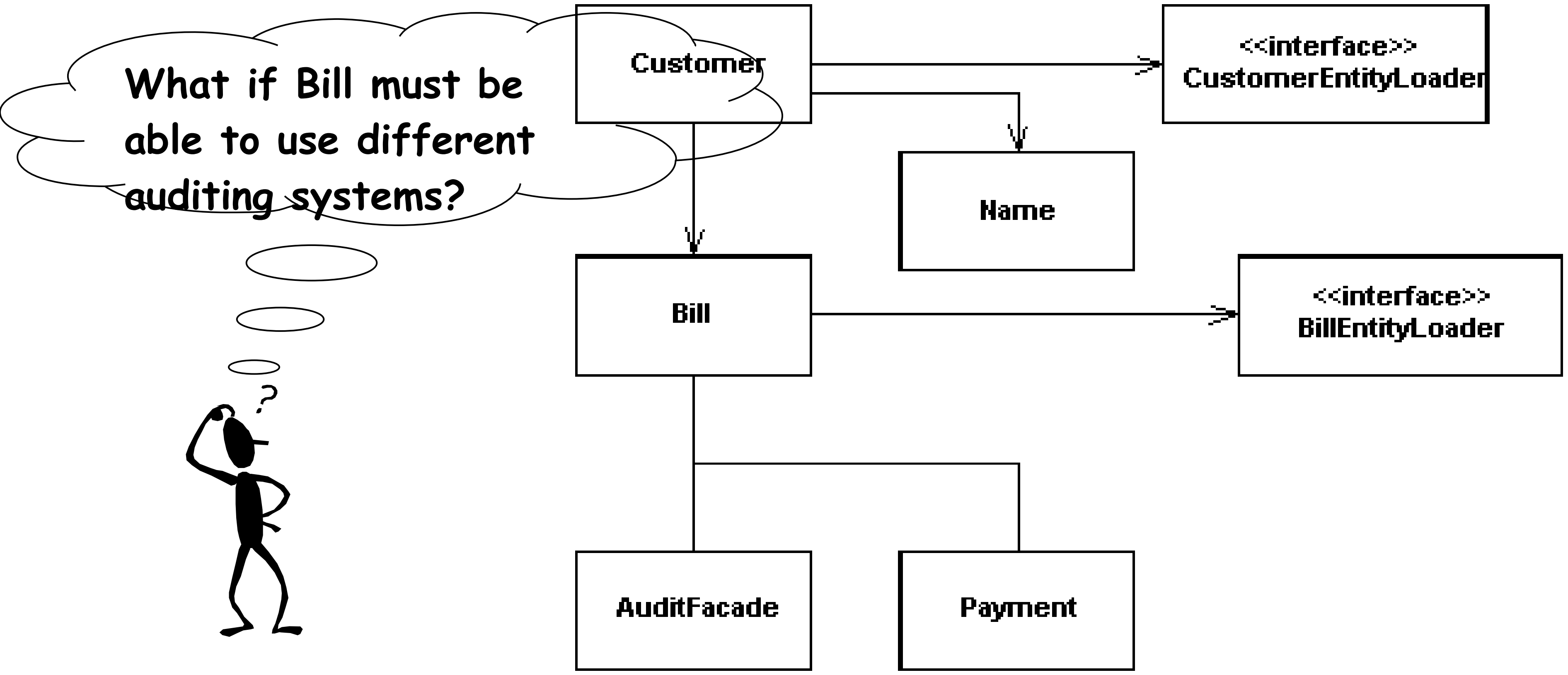
```
package client;
import service.*;
public class Client {
    Service service;
}
```

“Inject” the implementation into Client.
“Lookup” the implementation within Client.

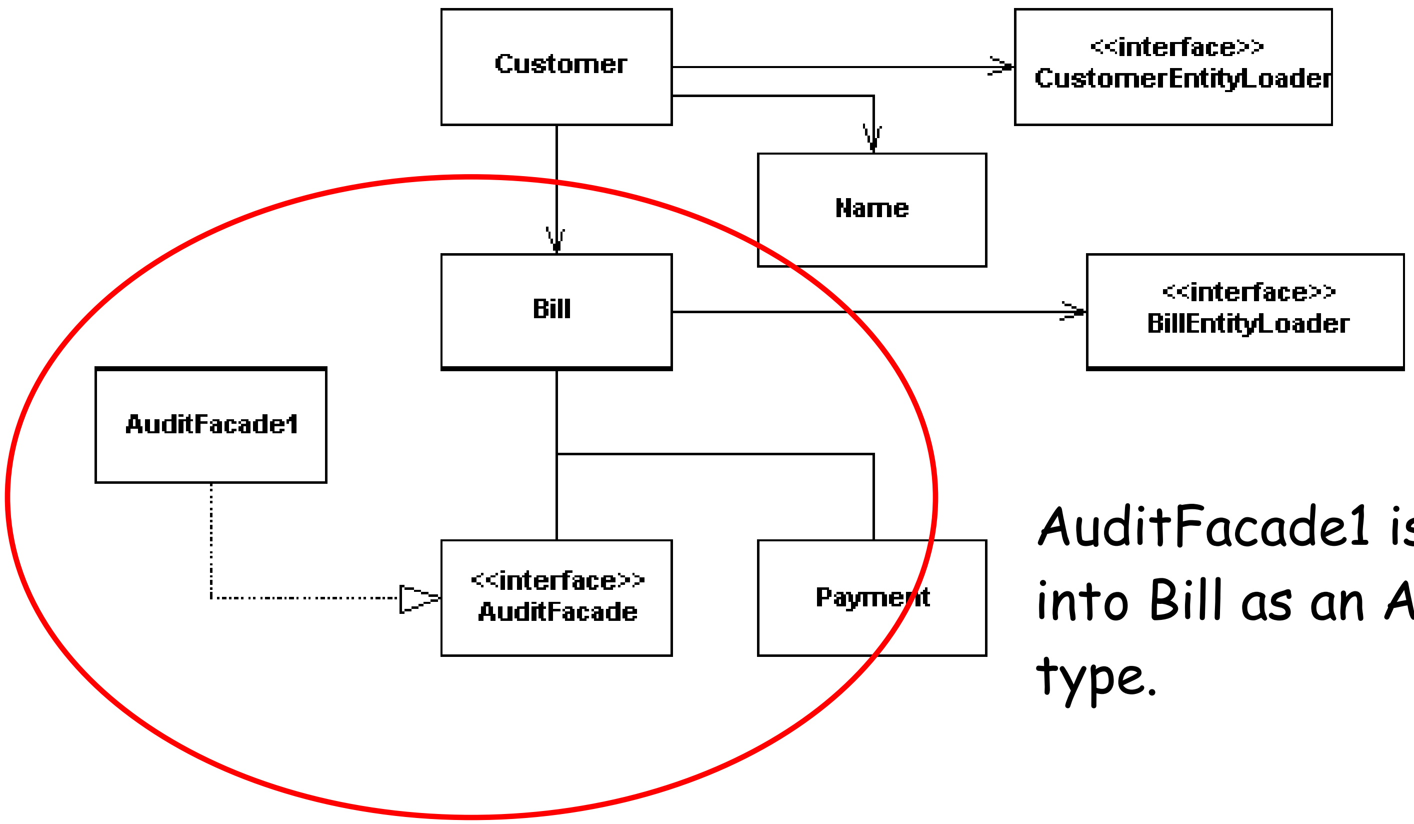
```
package service;
public interface Service {
  public void doService();

}
```

```
package service;
class ServiceImpl implements Service {
  public void doService() { ... }

}
```

Concrete Dependency

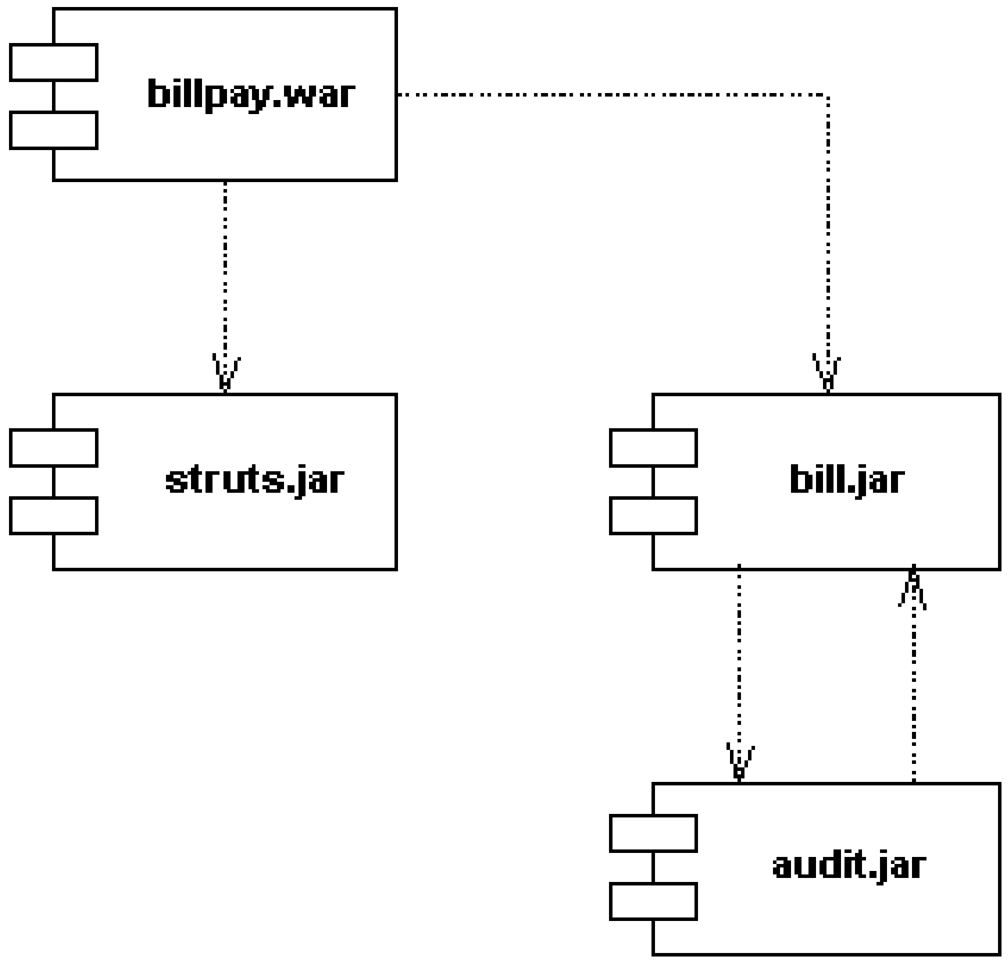

What if Bill must be able to use different auditing systems?
?
Customer
<<interface>> CustomerEntityLoader
Name
Bill
<<interface>> BillEntityLoader
AuditFacade
Payment

Abstract Dependency


Customer
<<interface>> CustomerEntityLoader
Name
Bill
<<interface>> BillEntityLoader
AuditFacade1
<<interface>> AuditFacade
Payment

AuditFacade1 is injected into Bill as an AuditFacade type.

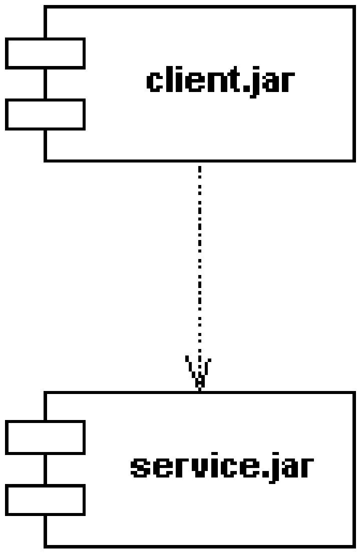
Component Relationships



AcyclicRelationships

- "Component relationships must be acyclic."
- A cyclic relationship exists when you can trace your dependencies, and end where you started.
- Cycles tend to creep into a system unknowingly.

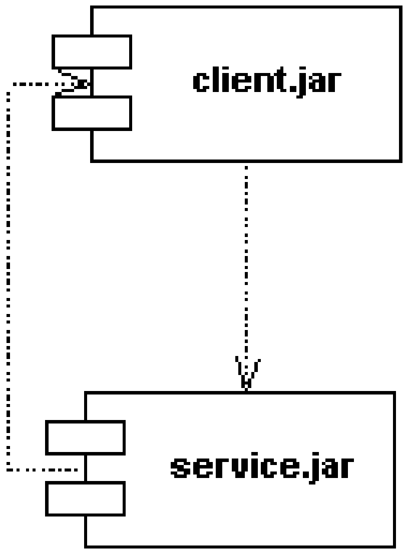
Cyclic and Acyclic Dependencies



```
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

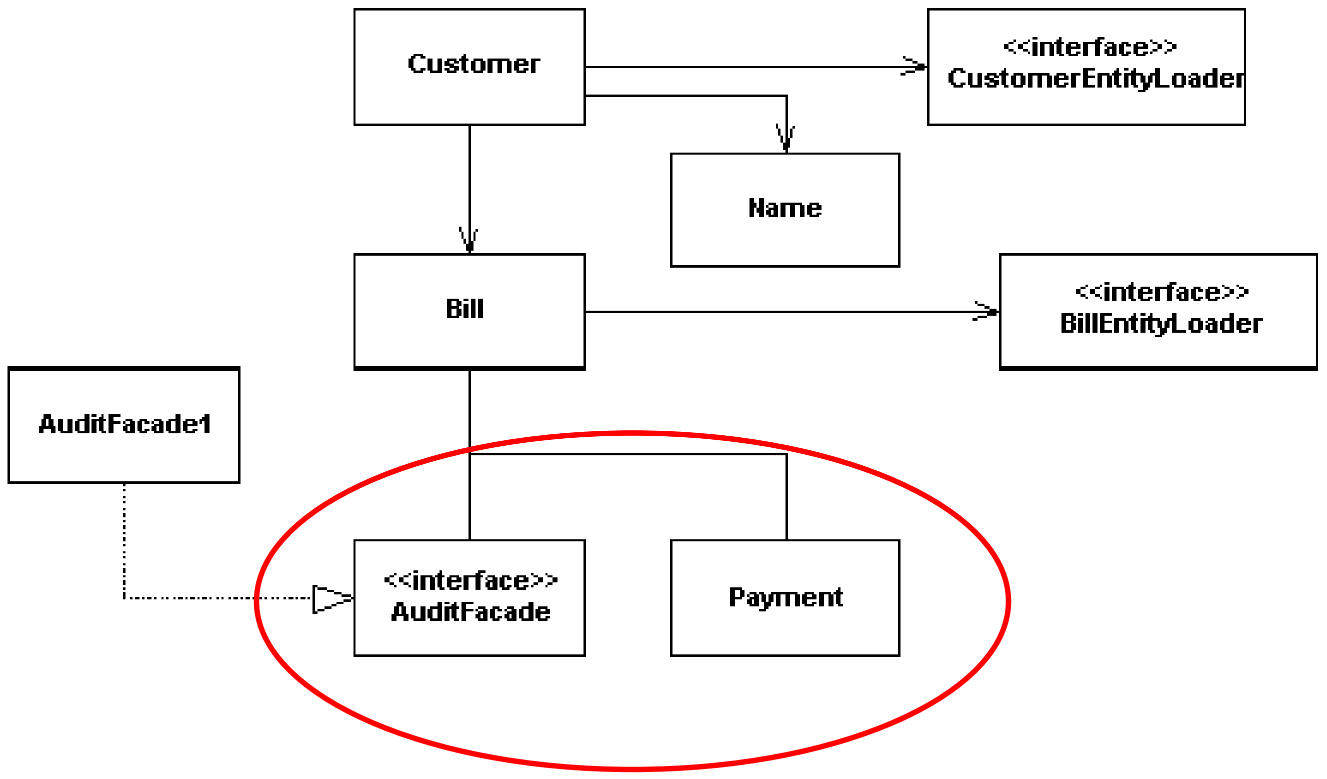
Leveling Relationships

- 0 are leaf component
  - 3rd party components such as struts, spring, hibernate, etc.
- 1 are lowest level components independent of anything else or only leaf components.
- n level components dependent on n-1 level components.
- Can only be done if relationships are acyclic

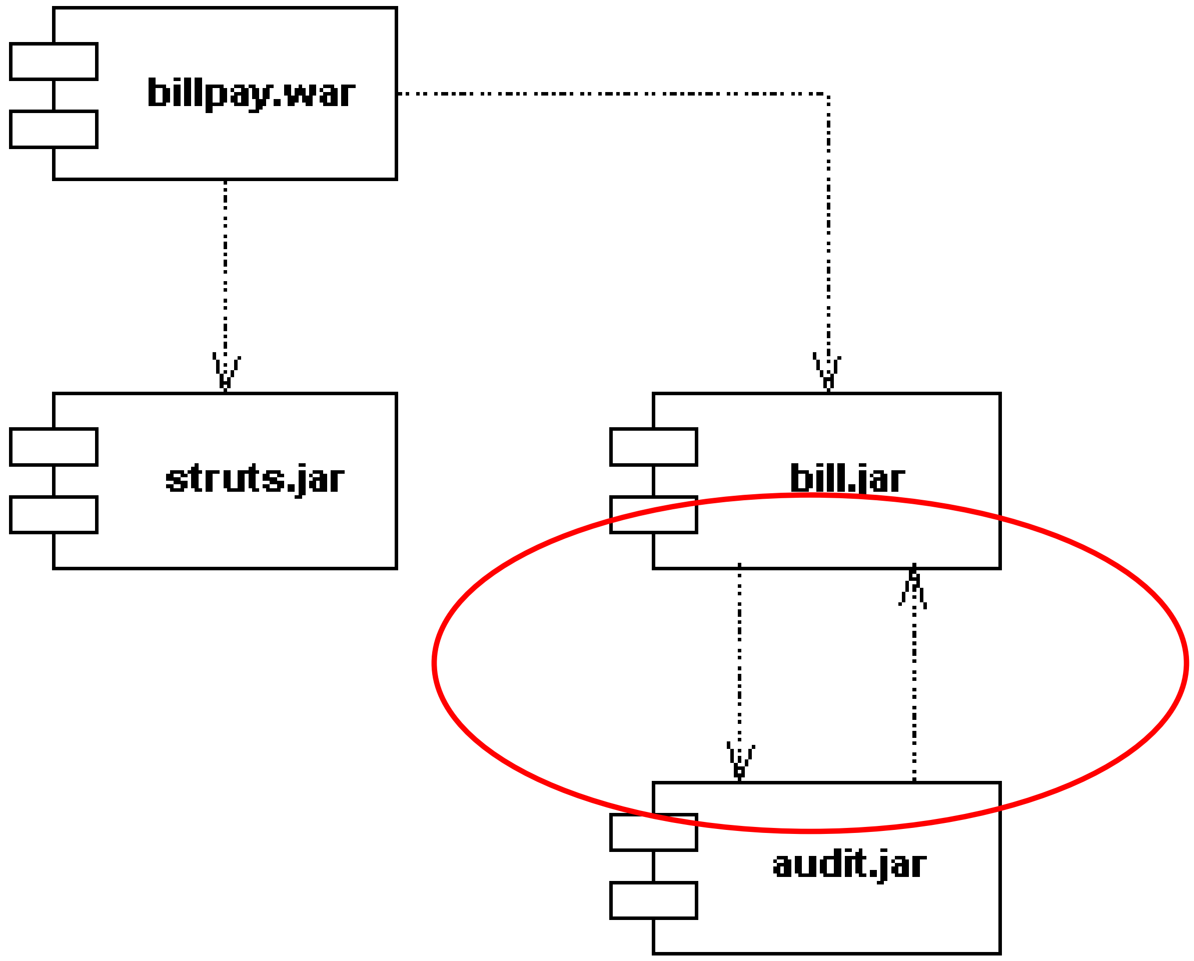
Notes

- Cycles can be broken
  - Escalation, Demotion, Callback
- The lower the level, the less volatile it must be.
  - Less volatile ➔ more abstract
- Levelized components can be effectively (and independently) tested.
- Levelized components can be built in order from 1 to n → LevelizedBuild

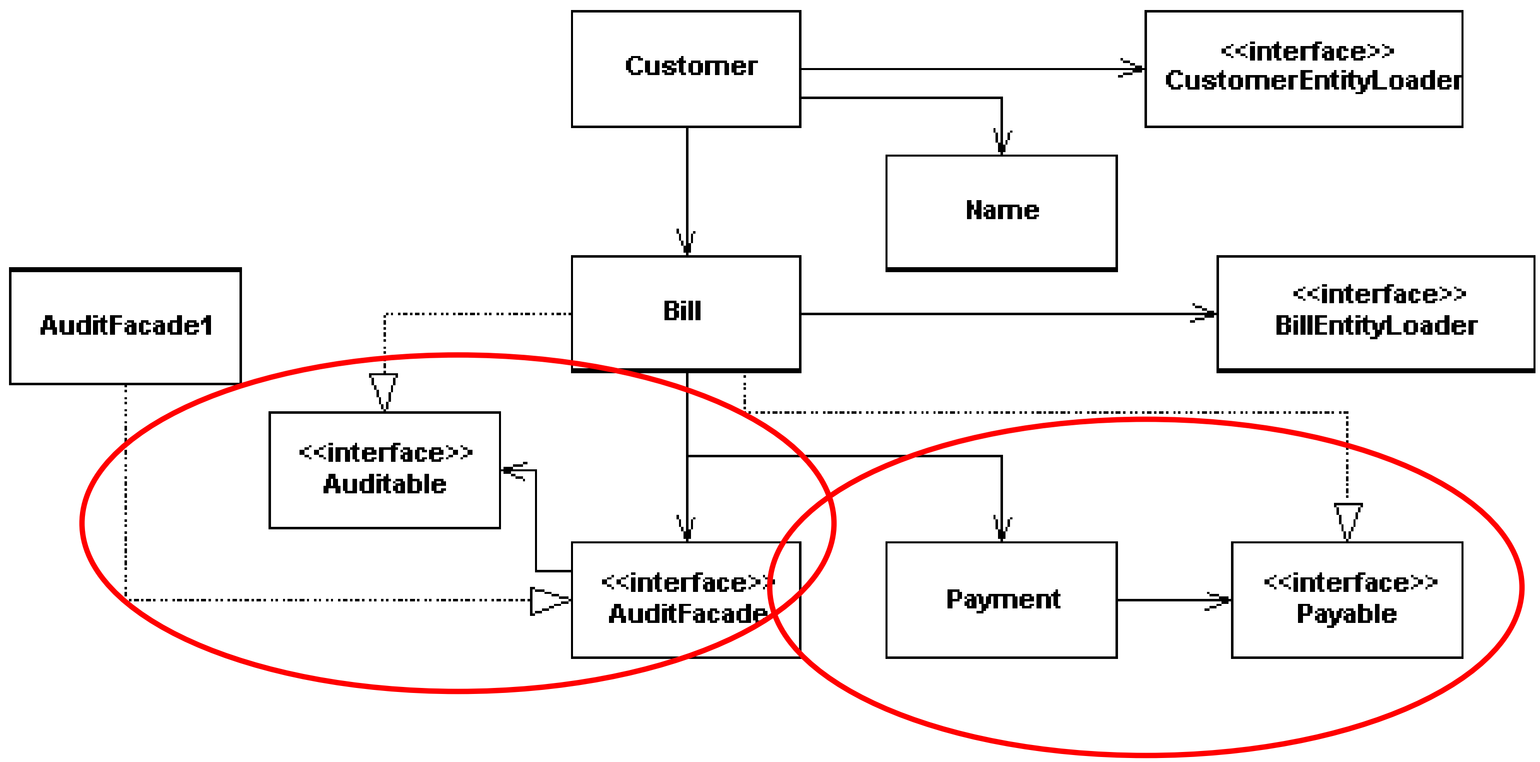
Recall - Abstract Dependency



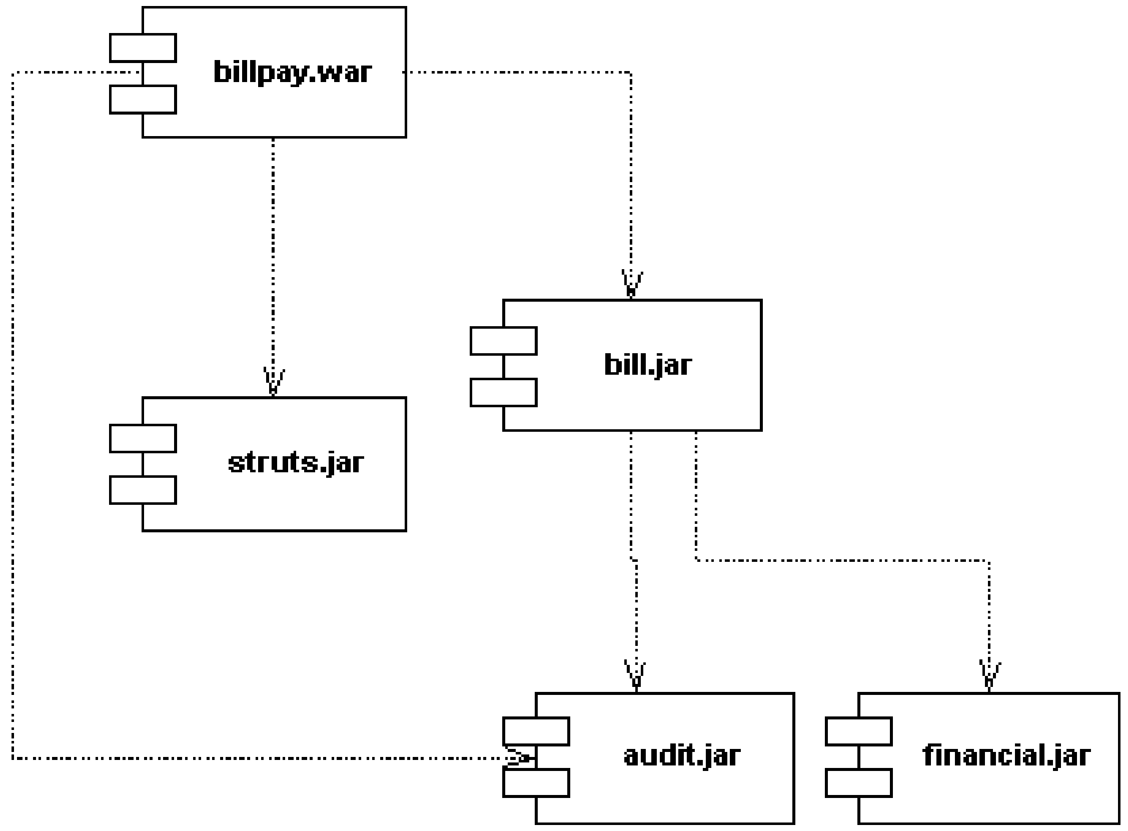
Recall - Component Relationships



Acyclic Relationships


Customer
<<interface>> CustomerEntityLoader
Name
AuditFacade1
Bill
<<interface>> BillEntityLoader
<<interface>> Auditable
<<interface>> AuditFacade
Payment
<<interface>> Payable

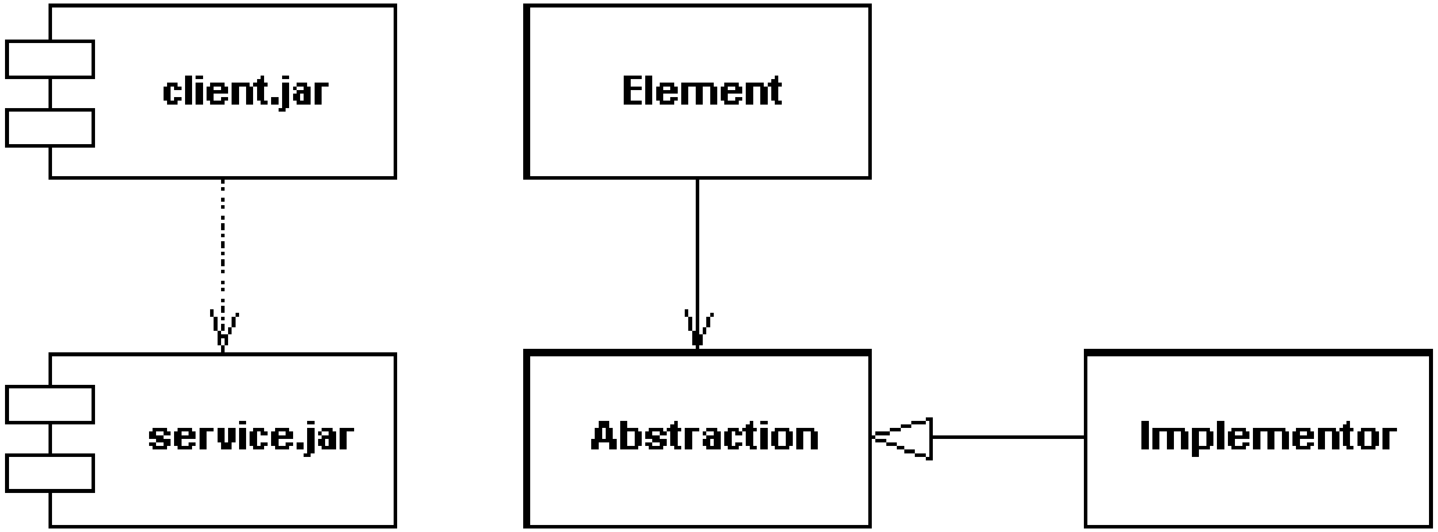
Levelized BillPay



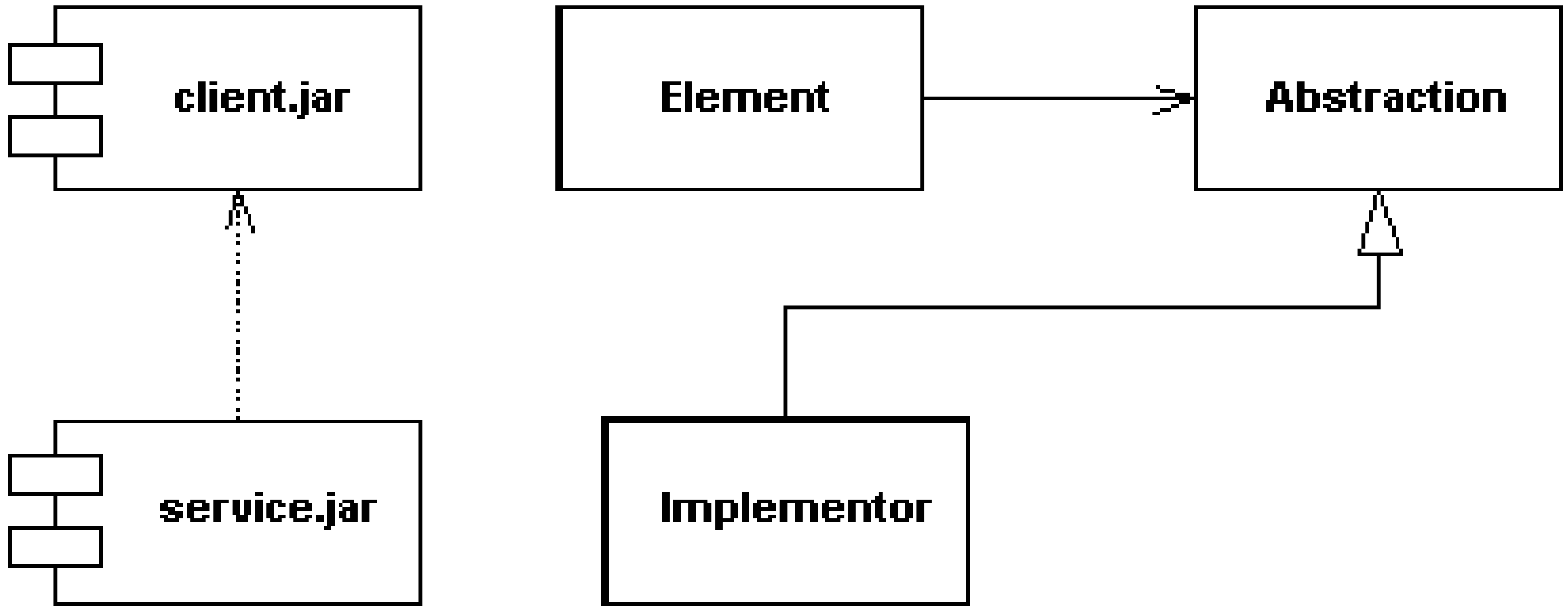
SeparateAbstractions

- “Separate abstractions from the classes that realize them.”
- Directed Dependency
  - Collocate abstraction and implementation
- Inverted Dependency
  - Collocate abstraction and referencing class
- Eliminated Dependency
  - Move abstraction to separate component

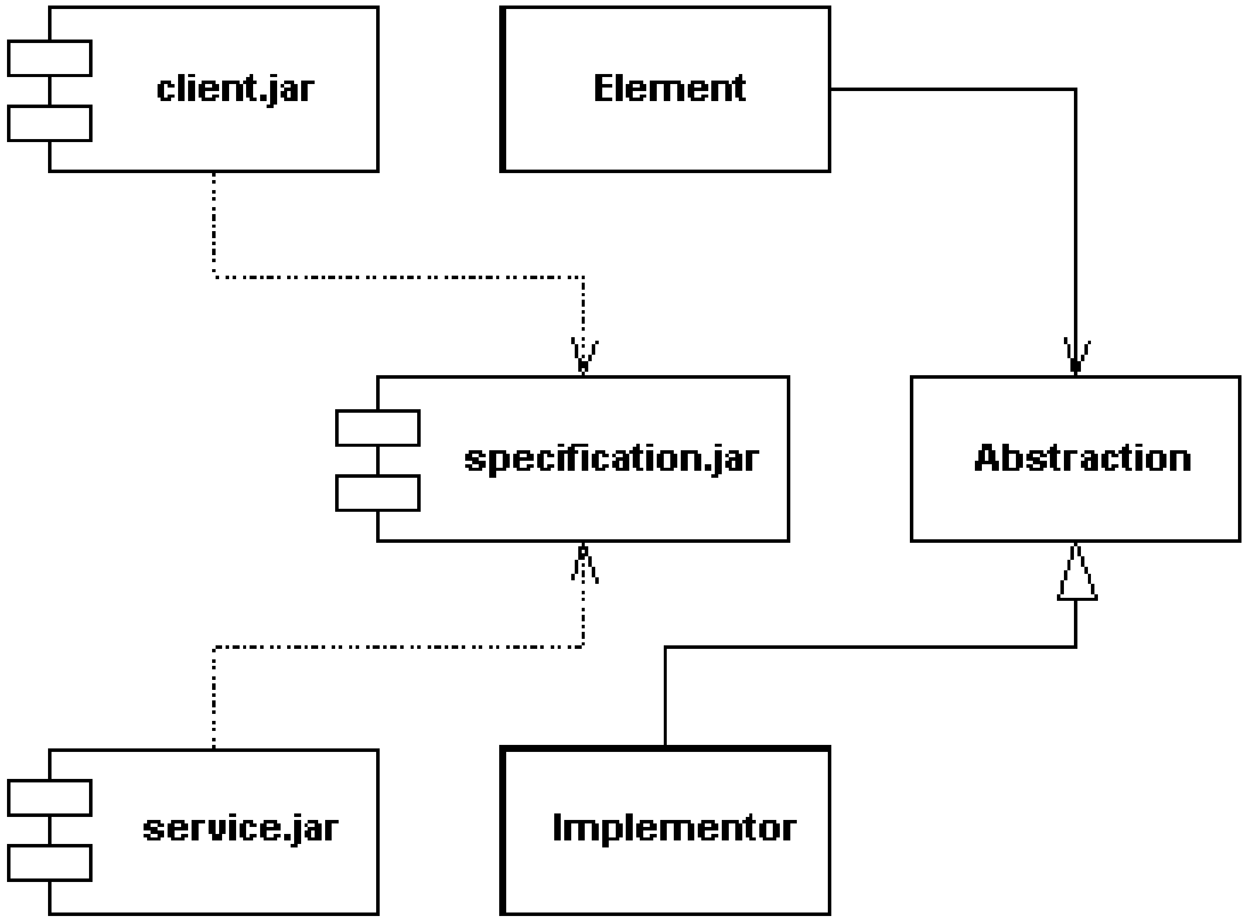
Direct Dependency



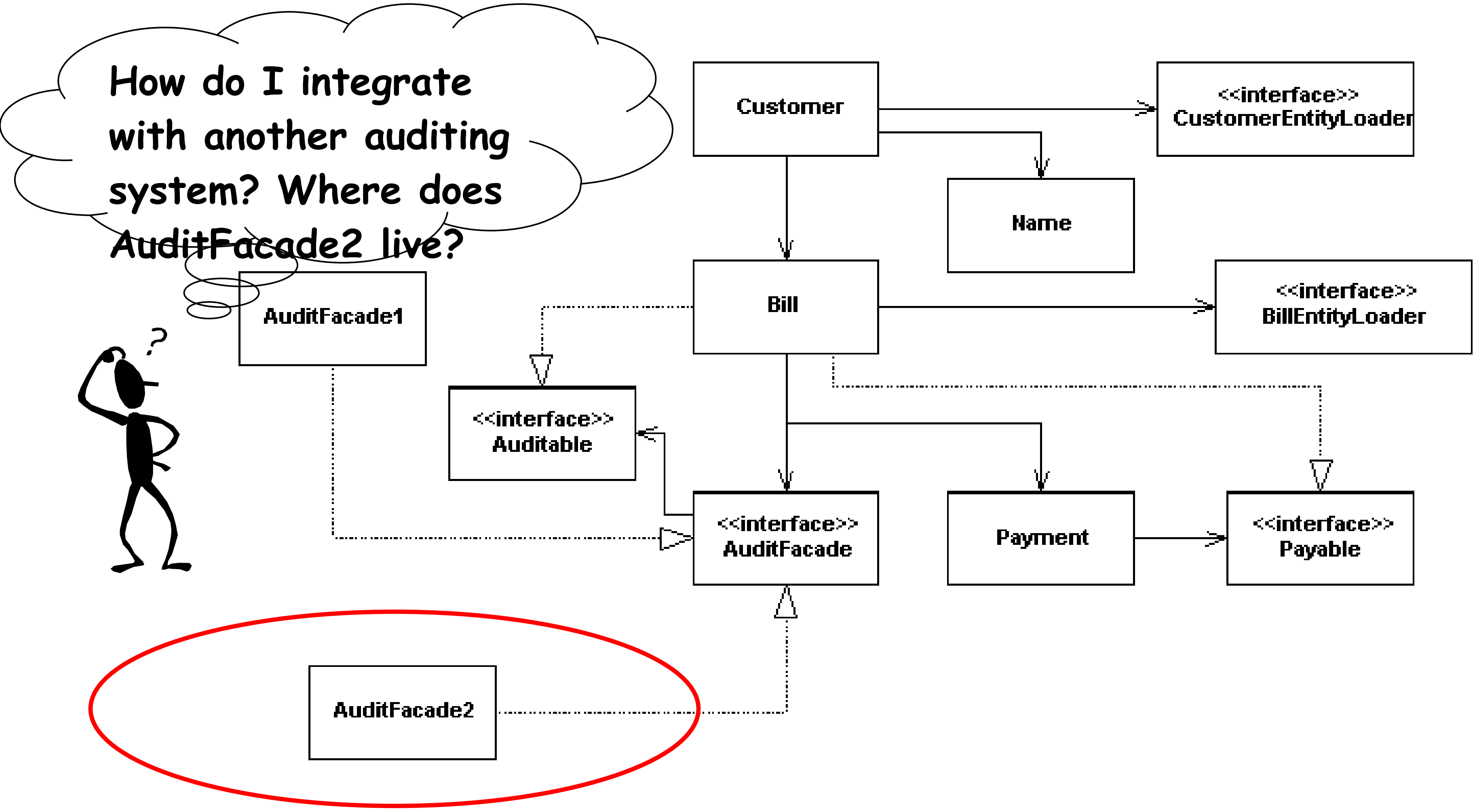
Inverted Dependency



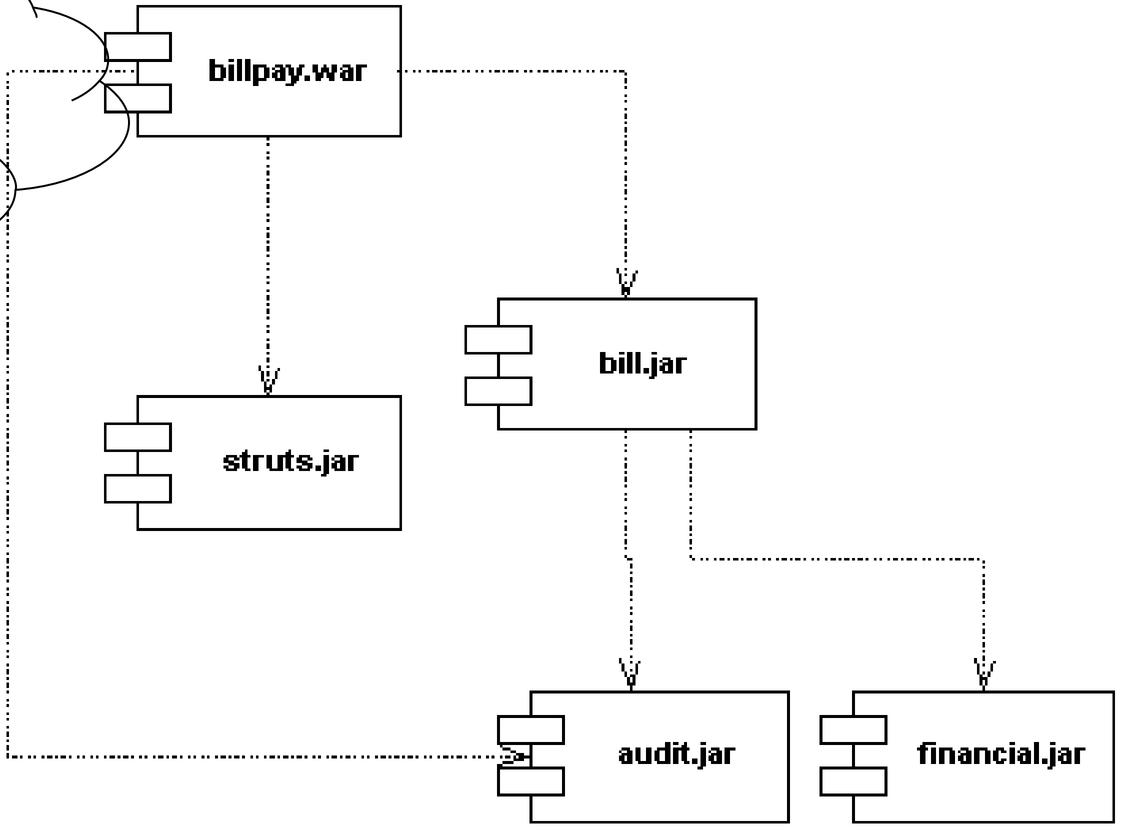
Eliminated Dependency



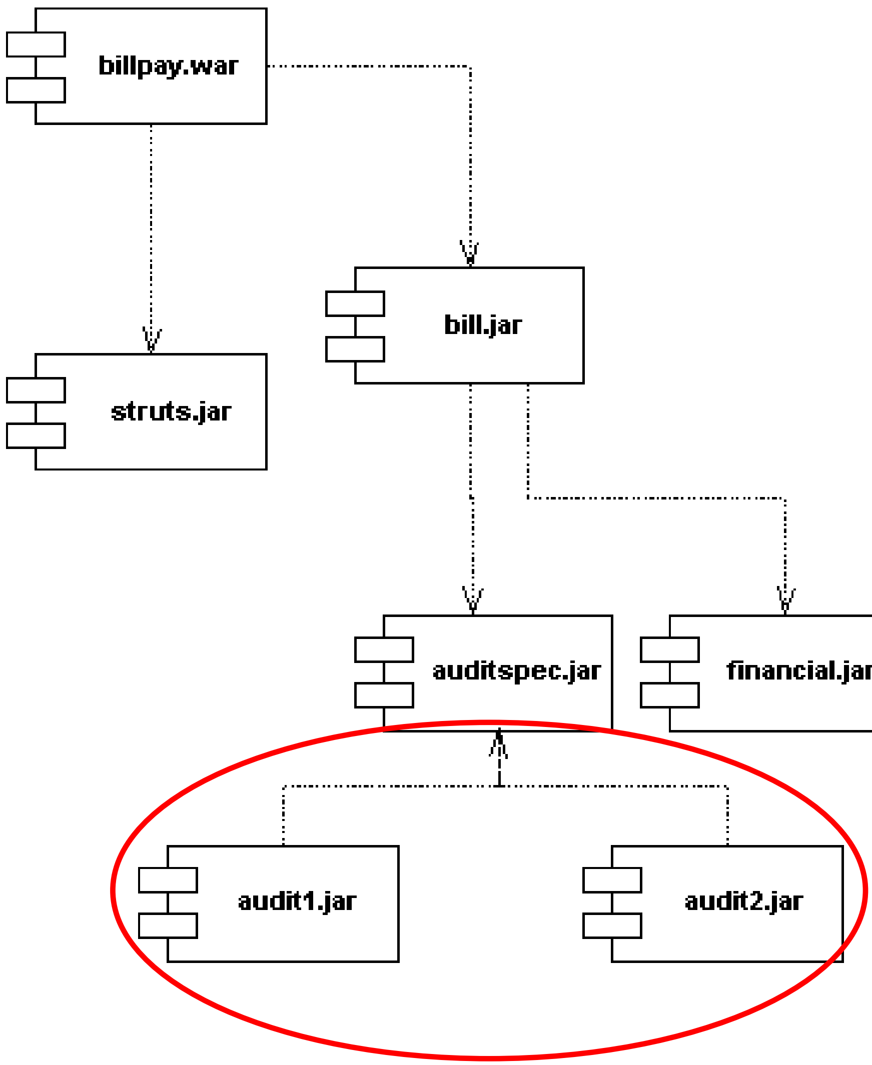
Acyclic Relationships


How do I integrate with another auditing system? Where does AuditFacade2 live?
Customer
<<interface>> CustomerEntityLoader
Name
Bill
<<interface>> BillEntityLoader
AuditFacade1
<<interface>> Auditable
<<interface>> AuditFacade
Payment
<<interface>> Payable
AuditFacade2

Recall - Levelized BillPay



Abstract Components

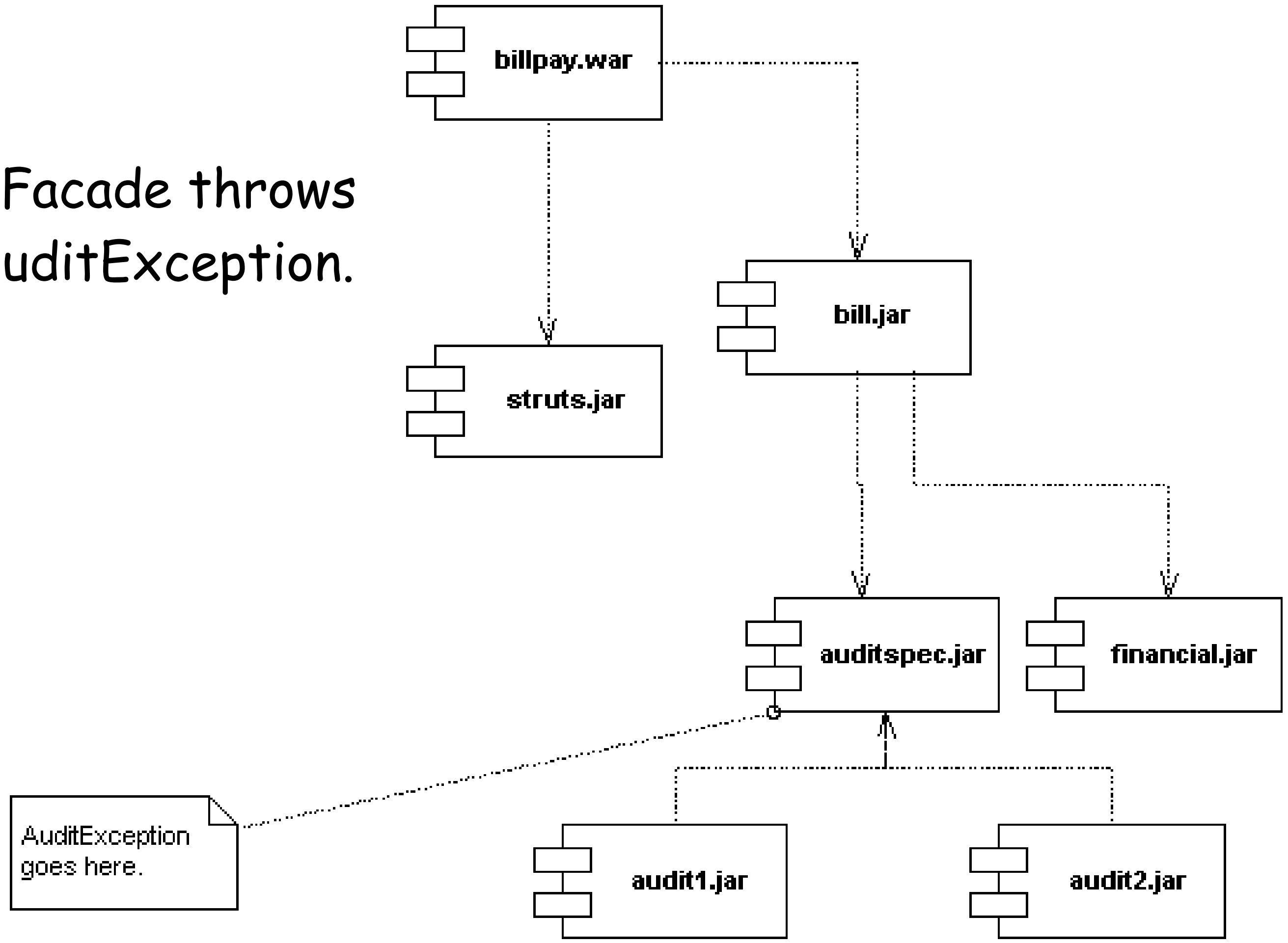


CollocateExceptions

- “Exceptions should be close to the classes that throw them.”
- Exceptions are often an afterthought.
- Consider using only unchecked exeptions.
  - If you decide to change, you won’t break everything.

Exception Placement

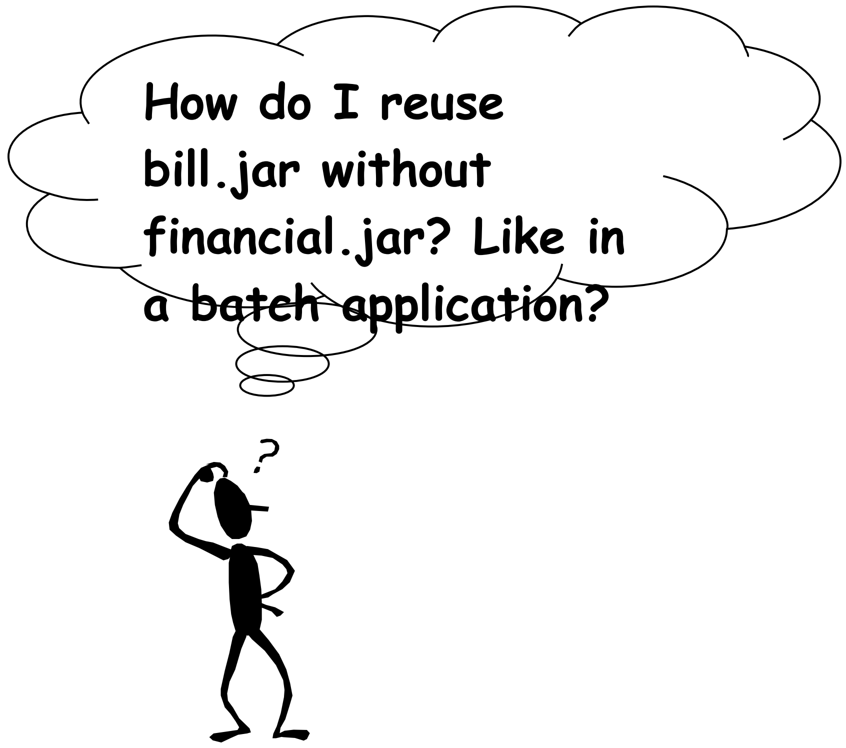
AuditFacade throws the AuditException.

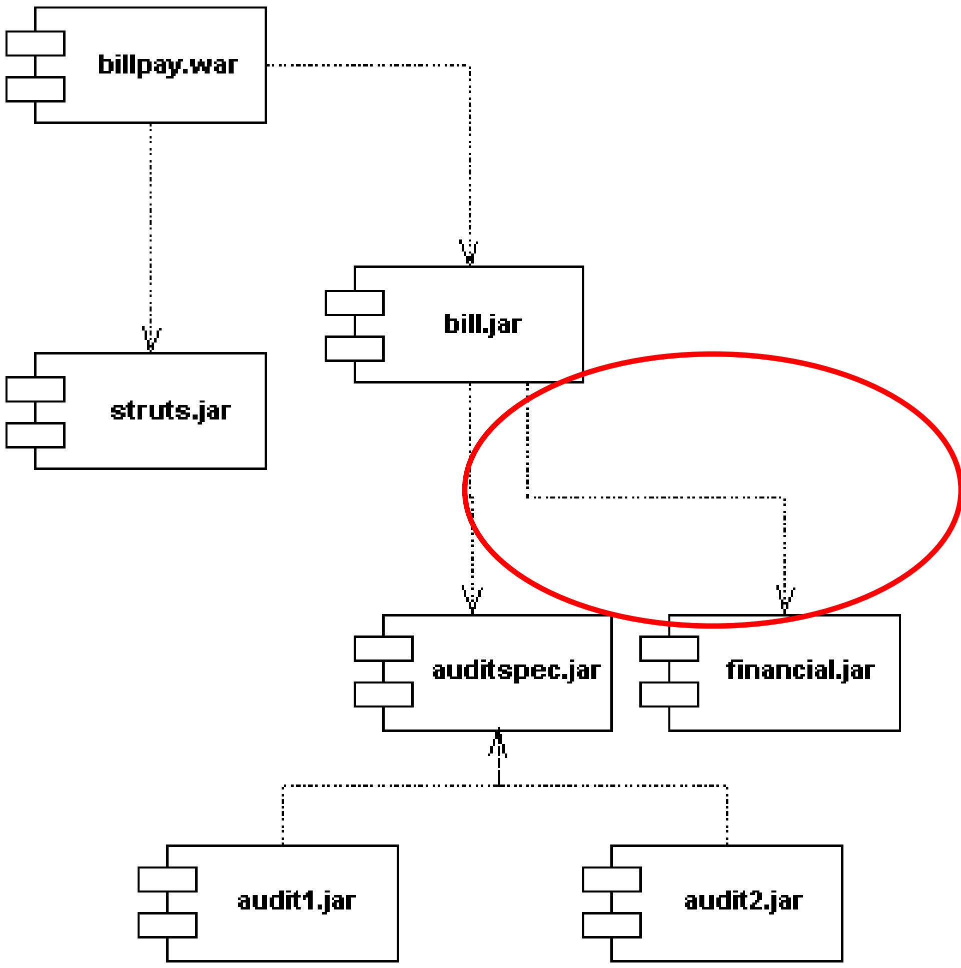


IndependentDeployment

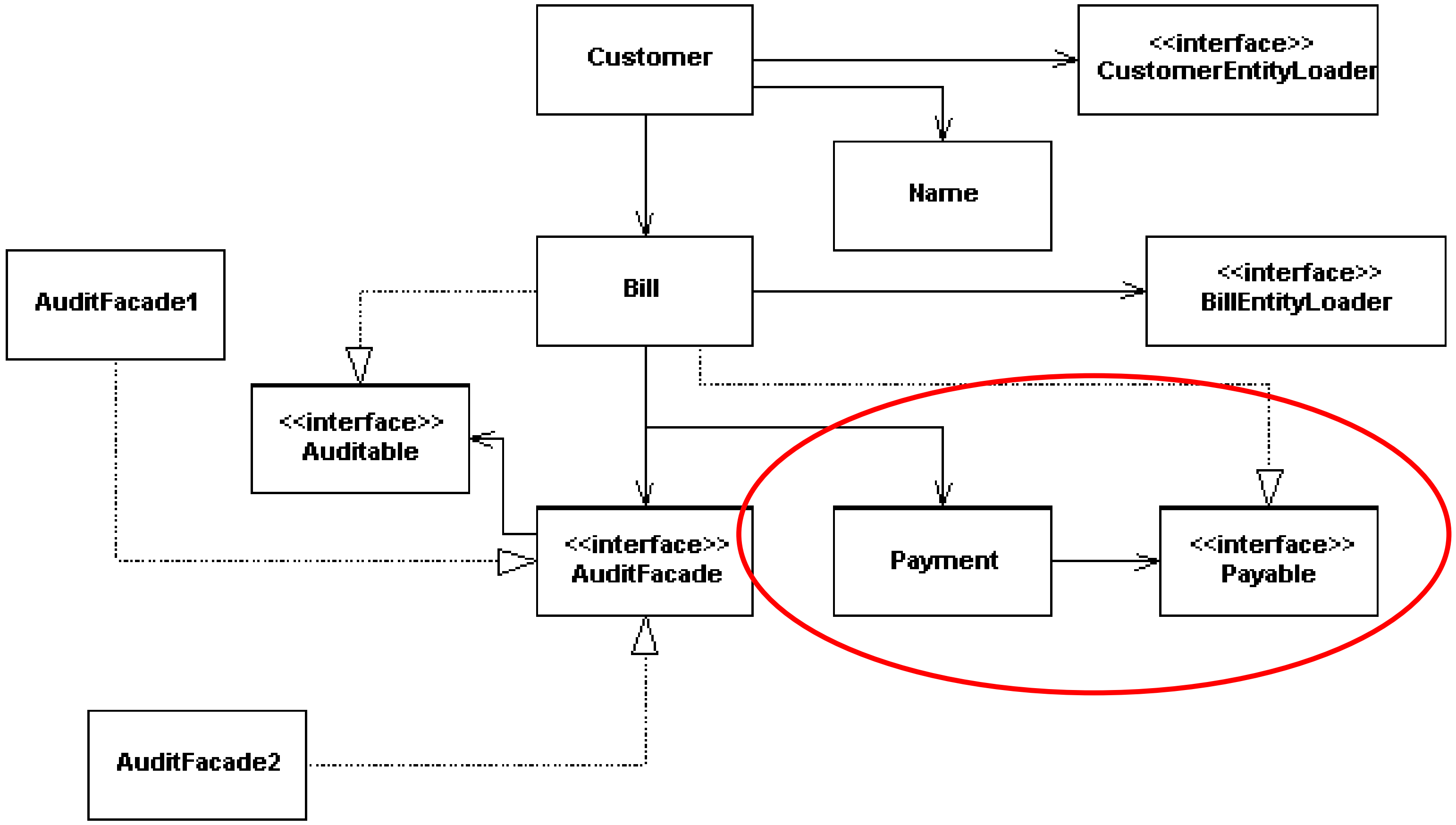
- "Components should be as independently deployable as possible."
- Minimize a component's outgoing dependencies.
- "Wire" components together
- Don't depend on the container
  - Reduces reuse
  - J2EE dependencies

Recall - Abstract Components

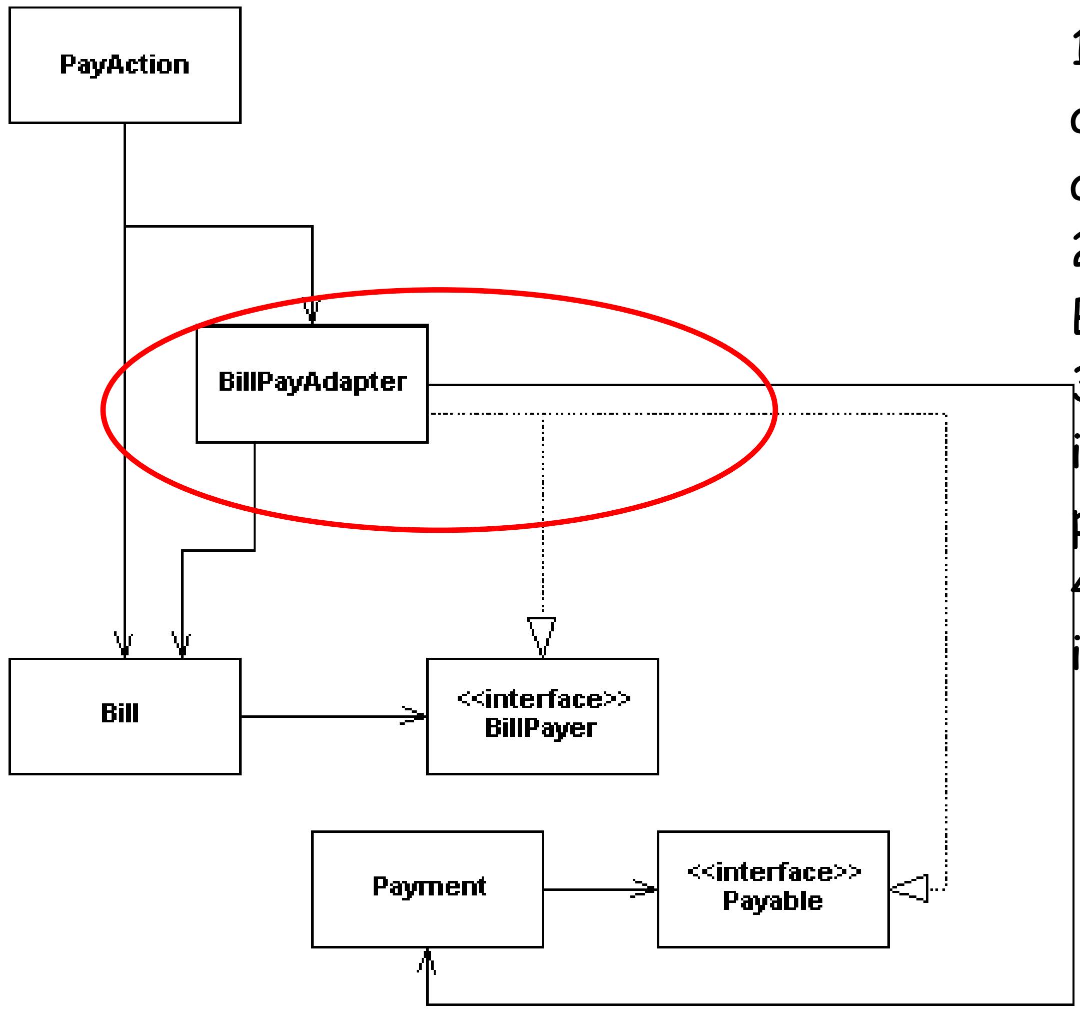

How do I reuse bill.jar without financial.jar? Like in a batch application?
?



Recall – Class Structure

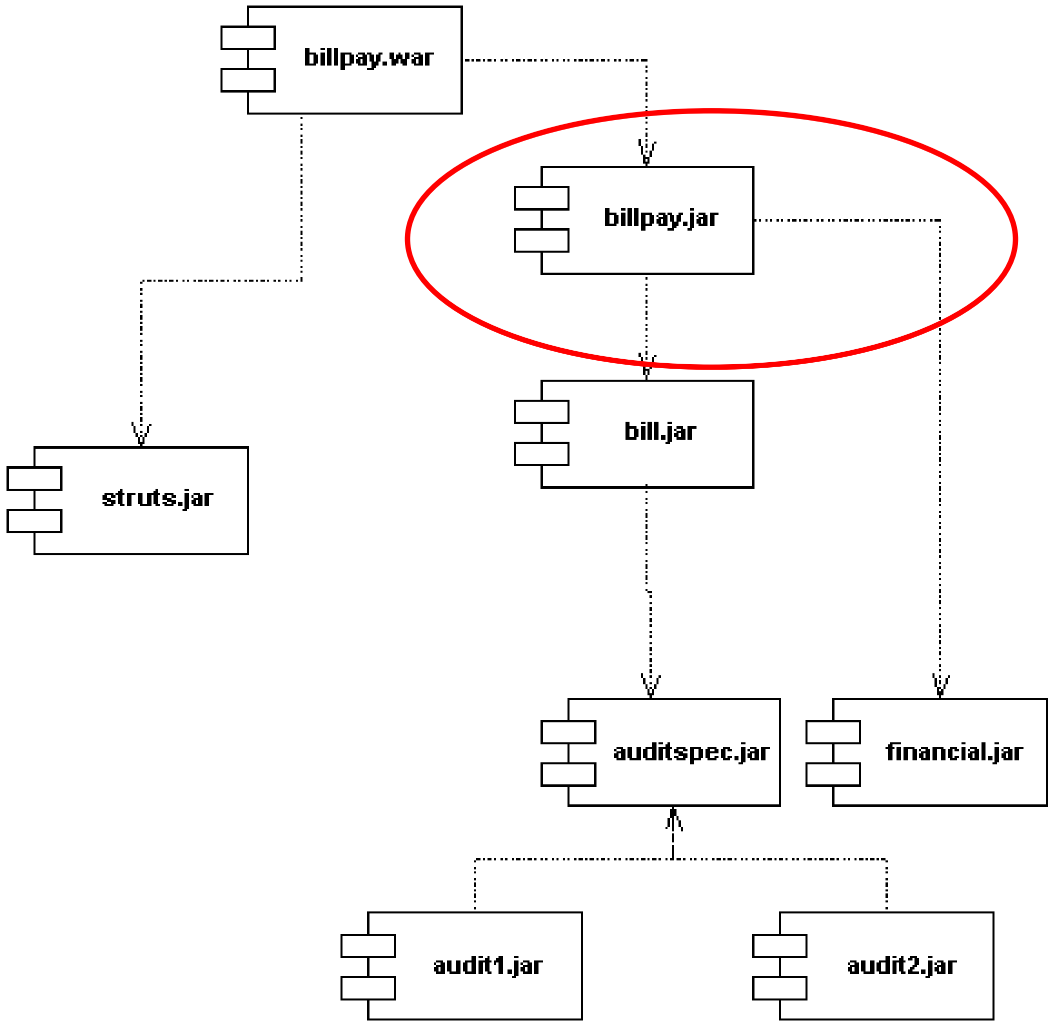


Class Structure



1.) PayAction invokes Bill.pay() and passes BillPayAdapter as a BillPayer.
2.) Bill.pay() invokes BillPayer.generateDraft()
3.)BillPayAdapeter.generateDraft() invokes Payment.generateDraft() passing itself as a Payable.
4.) Payment.generateDraft() invokes Payable.getAmount()

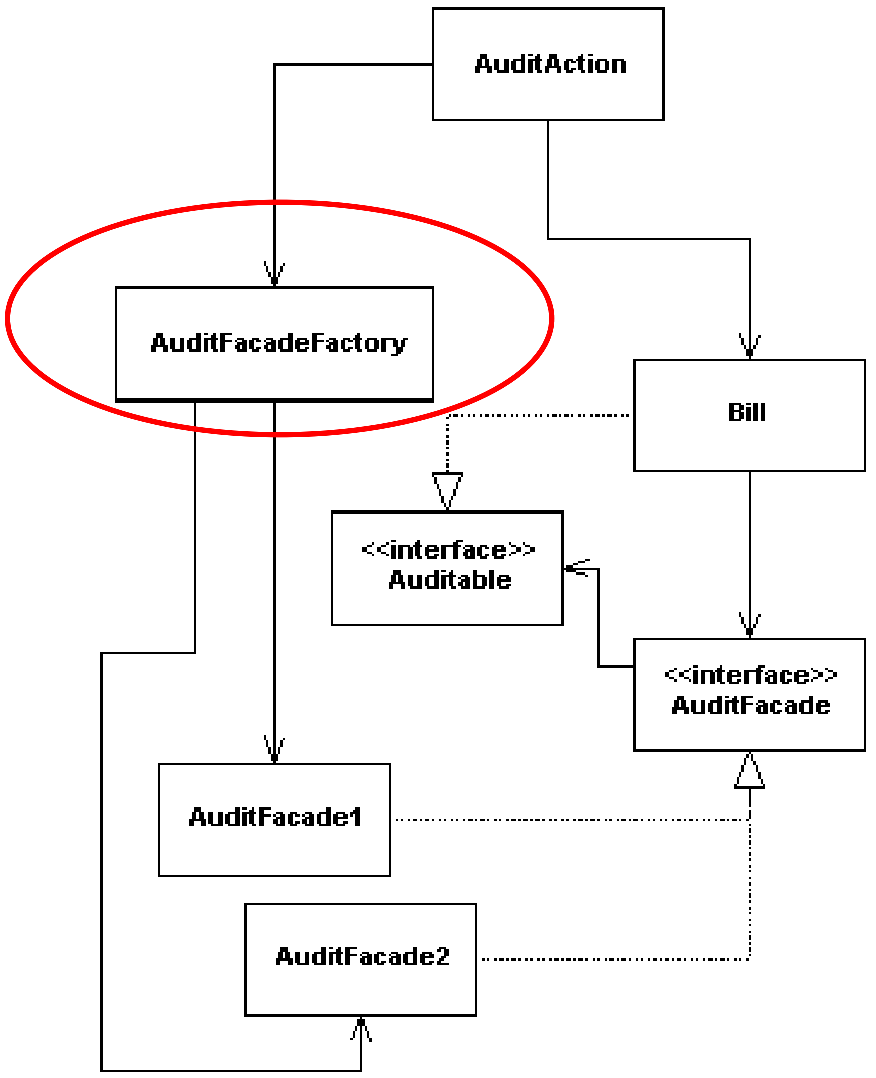
Reusing bill.jar



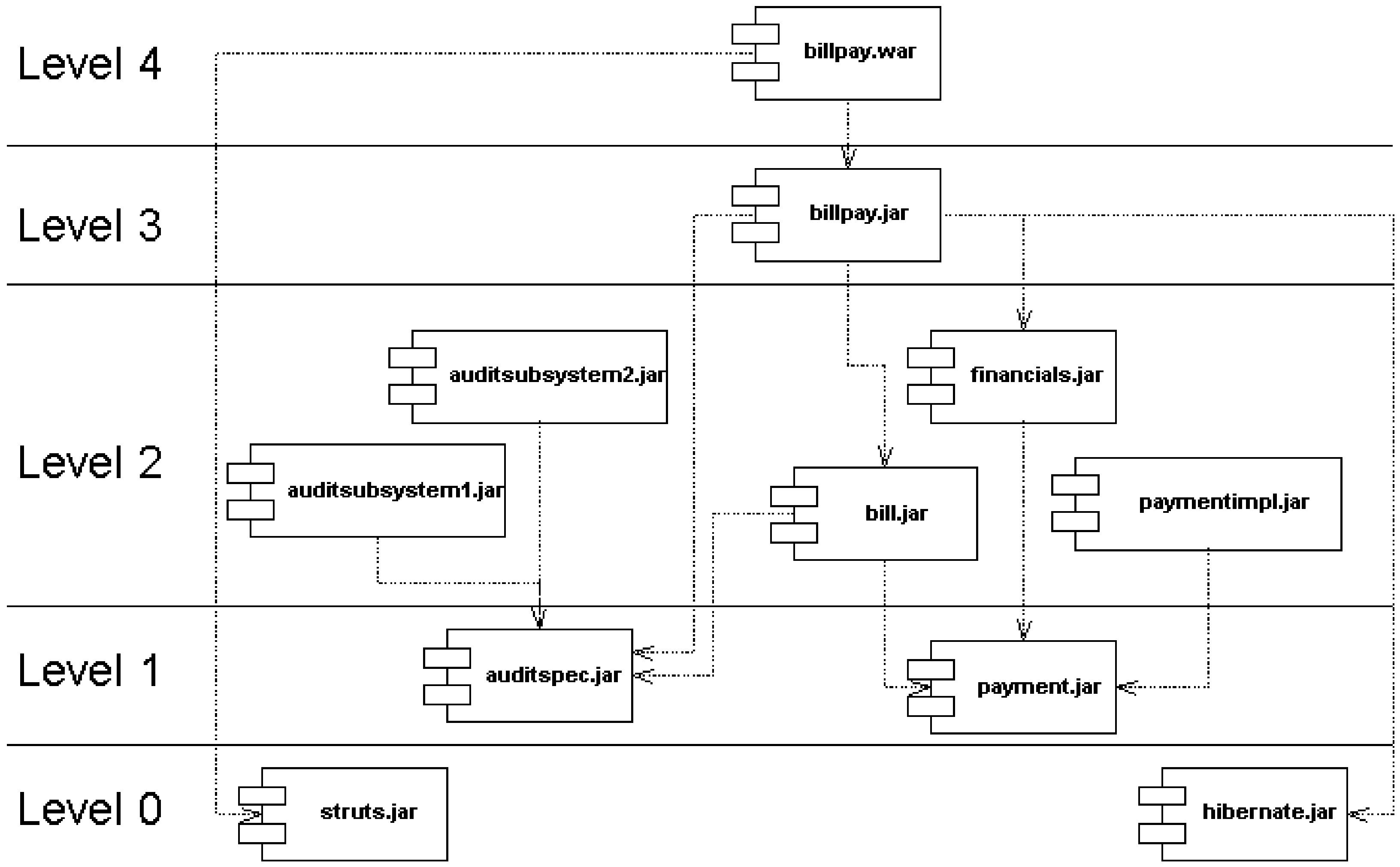
ImplementationFactory

- "Use factories to create a component's implementation."
- new violates AbstractDependency
  - Manage carefully

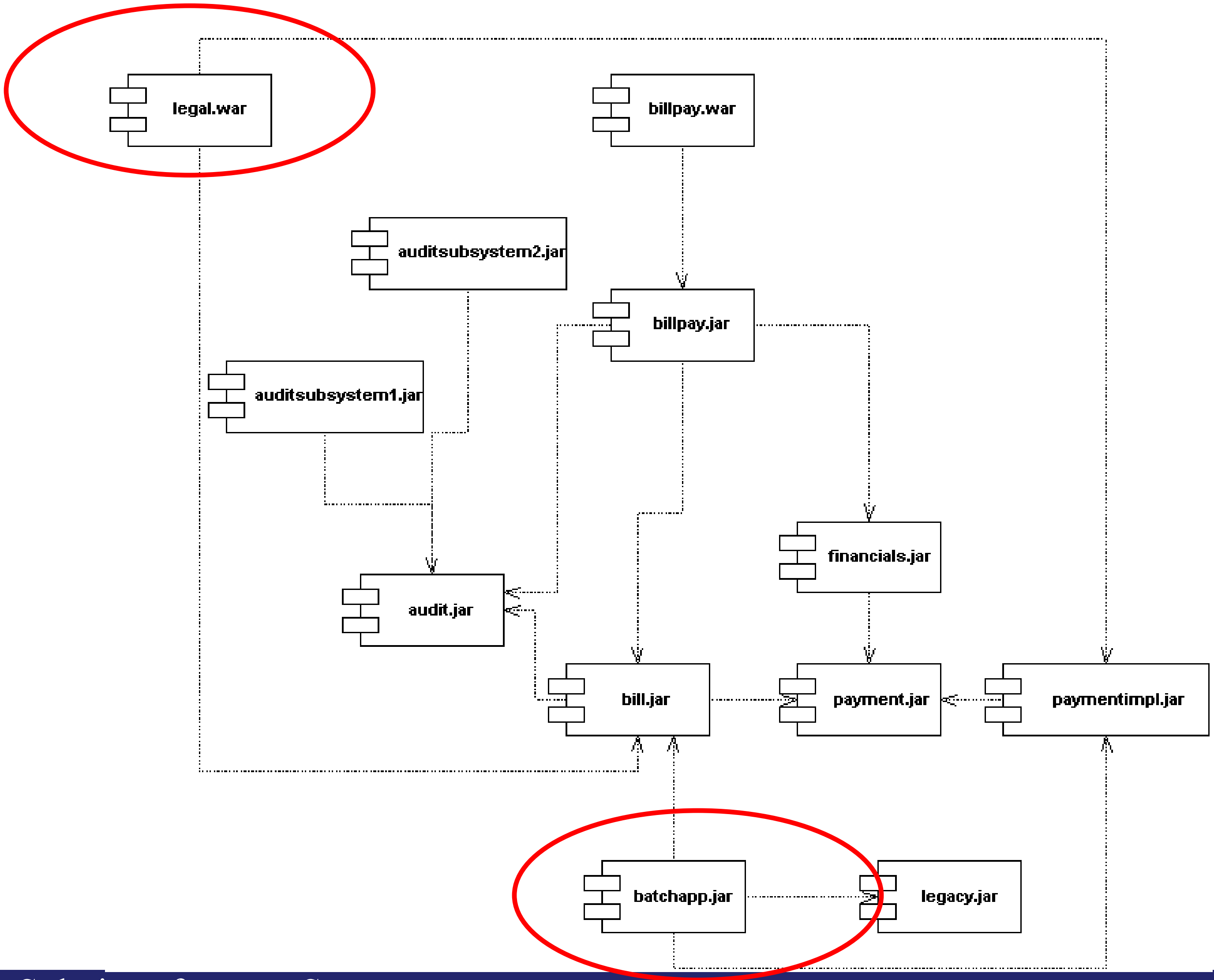
Factory Class



Final Structure



Extension


legal.war
billpay.war
auditsubsystem2.jar
billpay.jar
auditsubsystem1.jar
financials.jar
audit.jar
bill.jar
payment.jar
paymentimpl.jar
batchapp.jar
legacy.jar

Analyzing Jar Files

- Run Jar Analyzer
  - Generates xml showing dependencies between .jar files.
  - Ant task available to run as part of build process.
  - Feedback? Contributions?
  - Available at www.kirkk.com

Additional Resources

- www.kirkk.com
  - JarAnalyzer download and general information on software development.
- www.qwantify.com
  - Whitepapers, articles, and blogs on a variety of technical topics.
- www.extensiblejava.com
  - Resource devoted exclusively to dependency management.

Please complete your session evaluation forms