



Aspect-Oriented Programming

with *dynaop*

Bob Lee

crazybob@crazybob.org



Aspect-Oriented Programming

- 🍯 AOP for short
- 🍯 Improves upon OOP
 - 🍯 Code reuse
 - 🍯 Decomposition
 - 🍯 Dependency reduction
- 🍯 Modularizes *crosscutting concerns...*

Crosscutting Concerns

- ❏ Can't isolate to a single class
- ❏ Example: trace logging
- ❏ Analogy: Date is to class as trace logging is to aspect
- ❏ More difficult to spot than “cut & paste” repetition

```
class Foo {  
  
    Logger logger = ...;  
  
    void foo(String s) {  
        logger.log("enter foo: " + s);  
        // logic;  
        logger.log("exit foo");  
    }  
  
    int foo(int i) {  
        logger.log("enter foo: " + i);  
        // logic;  
        logger.log("exit foo: " + result);  
        return result;  
    }  
}  
  
class Bar {  
  
    Logger logger = ...;  
  
    void bar(String s) {  
        logger.log("enter bar: " + s);  
        // logic;  
        logger.log("exit bar");  
    }  
}
```

More Examples

- 🔸 Thread Synchronization
- 🔸 Transactions
- 🔸 Security
- 🔸 Performance optimization
- 🔸 Error handling
- 🔸 Logging, debugging, metrics
- 🔸 Business Logic



The Framework: dynaop

- ✿ <http://dynaop.dev.java.net/>
- ✿ In the Java spirit
- ✿ EJB meets AspectJ
- ✿ Proxy design pattern
- ✿ Development tools
- ✿ Plays nicely with J2EE, applets, etc.
- ✿ BeanShell configuration

Aspect

- ❏ **Associates advice with pointcuts**
- ❏ **Advice**
 - ❏ Mixin Introduction
 - ❏ Interceptor
- ❏ **Pointcut**
- ❏ **Proxy**
 - ❏ An object plus all of its aspects
 - ❏ Analogy: object is to atom as proxy is to molecule

Mixin Introduction

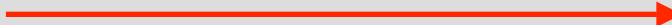
- Combines two objects
- Mixin* for short
- Declarative inheritance
- Implemented as a POJO

```
interface Identifiable {  
  
    long getId();  
    void setId(long id);  
}  
  
class IdentifiableMixin  
    implements Identifiable {  
  
    long id = -1;  
  
    public long getId() {  
        return this.id;  
    }  
  
    public void setId(long id) {  
        this.id = id;  
    }  
  
    ...  
}
```

Interceptor



- ❏ **Intercepts method invocations**
- ❏ **Executes code before and after invocation**

```
class TraceInterceptor implements Interceptor {  
  
    public Object intercept(Invocation i)  
        throws Throwable {  
        Logger logger = ...;  
        logger.log("enter " + i.getMethod().getName()  
            + ": " + Arrays.asList(i.getArguments()));  
  
        Object result = i.proceed()    
  
        logger.log("exit " + i.getMethod().getName());  
        return result;  
    }  
}
```

```
class Bar {  
  
    void bar(String s) {  
        // logic;  
    }  
}
```

Interceptor Example



Pointcut

- ❏ Picks classes
- ❏ Combinable and reusable
 - ❏ Use a class pointcut to pick method return types
- ❏ Extensible
 - ❏ Pick methods based on annotations (JSR-175)



```
interceptor(  
    Bar.class,  
    "^void bar\\(",  
    new TraceInterceptor()  
);  
  
mixin(  
    Bar.class,  
    IdentifiableMixin.class  
);
```

```
class Bar {  
    void bar(String s) {  
        // logic;  
    }  
}
```

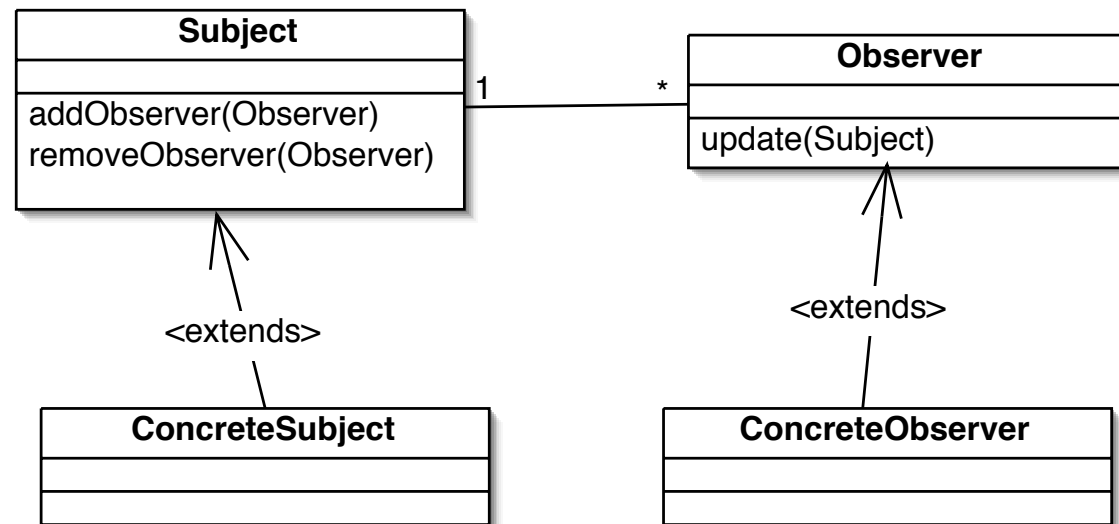
Pointcut Example

```
// look up proxy factory.  
ProxyFactory proxyFactory = ...;  
  
// create new instance of Bar using factory.  
Bar b = (Bar) proxyFactory.extend(Bar.class);  
  
// this will be traced.  
b.bar("Hello, World!");  
  
// this is implemented by IdentifiableMixin.  
((Identifiable) b).setId(100);
```

Usage

Example: Observer Design Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. [c2.com]
- Example implementation: `java.util.Observable`



The Subject

```
interface MyClass {  
    void doSomething();  
}  
  
class MyClassImpl implements MyClass {  
    void doSomething() {  
        // business logic  
    }  
}
```



The OO Way



Inheritance

```
class MyClassImpl implements MyClass
    extends SubjectImpl {

    void doSomething() {
        // business logic
        notifyObservers();
    }
}
```

Delegation

```
class MyClassImpl implements MyClass,  
    Subject {  
  
    Subject subject = ...;  
  
    void doSomething() {  
        // business logic  
        notifyObservers();  
    }  
  
    void notifyObservers() { subject.notifyObservers(this) }  
    void addObserver(Observer o) { subject.addObserver(o) }  
    void removeObserver(Observer o) { subject.removeObserver(o) }  
}
```

Proxy Pattern

```
class MyClassSubjectProxy implements MyClass extends SubjectImpl {  
  
    MyClass myClass;  
  
    MyClassSubjectProxy(MyClass myClass) { ... }  
  
    void doSomething() {  
        myClass.doSomething();  
        notifyObservers();  
    }  
}
```



The *dynaop* Way



Configuration

```
// attach SubjectMixin to instances of MyClass
mixin(
  MyClass.class,
  SubjectMixin.class
);

// apply interceptor to MyClass.doSomething()
interceptor(
  MyClass.class,
  "doSomething", // regexp
  new SubjectInterceptor()
);
```

Usage

```
// use default proxy factory
ProxyFactory proxyFactory = ProxyFactory.getInstance();

// proxy instance
MyClass proxied = (MyClass) proxyFactory.wrap(new MyClassImpl());

// add an observer
((Subject) proxied).addObserver(new MyObserver());

// do something.
proxied.doSomething();
```

Subject Mixin

```
public class SubjectMixin implements Subject, ProxyAware {  
  
    Collection observers = ...;  
    Subject subject;  
  
    public void addObserver(Observer observer) { ... }  
    public void notifyObservers() { ... }  
    public void removeObserver(Observer observer) { ... }  
  
    // this is called by dynaop.  
    public void setProxy(Proxy proxy) {  
        this.subject = (Subject) proxy;  
    }  
}
```

Subject Interceptor

```
public class SubjectInterceptor implements Interceptor {  
  
    public Object intercept(Invocation invocation) throws Throwable {  
        // proceed to the target method implementation.  
        Object result = invocation.proceed();  
  
        // invoke our SubjectMixin...  
        Subject subject = (Subject) invocation.getProxy();  
        subject.notifyObservers();  
  
        return result;  
    }  
}
```

Reuse





```
subjects = union(MyClass.class, YourClass.class);  
observedMethods = ...;
```

```
mixin(  
    subjects,  
    SubjectMixin.class  
);
```

```
interceptor(  
    subjects,  
    observedMethods,  
    new SubjectInterceptor()  
);
```

AOP vs. OOP

Pros

-  Less code, 100% reuse
-  We only need to hook once
-  Decouples classes from crosscutting concerns
-  Decouples crosscutting concerns from each other

Cons

-  Casting
-  Harder to troubleshoot



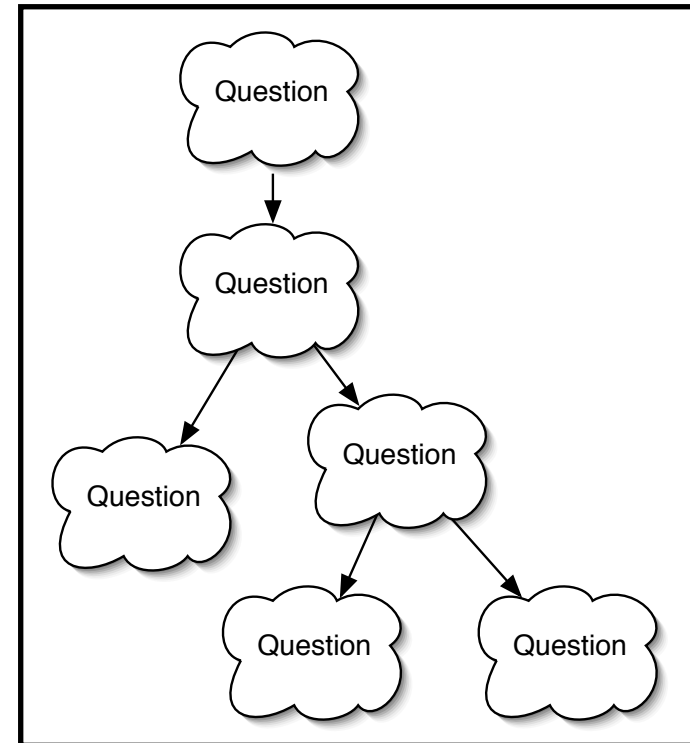
More Fun Example...

Patching Leaky Connections

- ❏ Per spec., JDBC Connections clean up after themselves
- ❏ In the real world, some connection pools and drivers don't

Undo Logic

- Web wizard
- Model
- Forms
- Transactions*



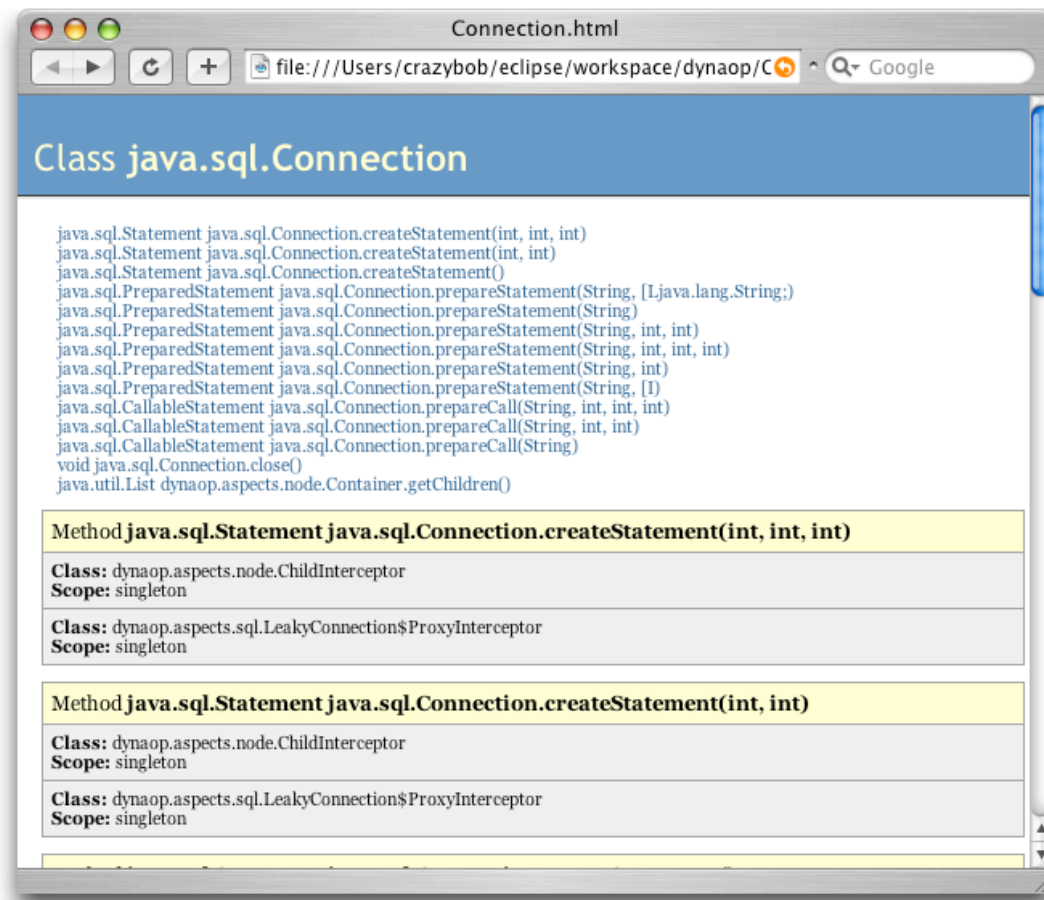
More Use Cases

- 🍯 Performance optimization
 - 🍯 Lazy loading
 - 🍯 Loading policies...
- 🍯 Command Pattern
 - 🍯 Not needed as much if at all

More Features




- ❏ **Object serialization**
- ❏ **AOP Alliance**
- ❏ **Binary web service**
- ❏ **Custom pointcuts and advice factories**
- ❏ **Dispatch interceptor**
- ❏ **High performance**

Tool Support





Coming Soon...

-  **Abstract advice**
-  **Transparent proxies**
-  **Invocation Pointcuts**

Recommended Reading

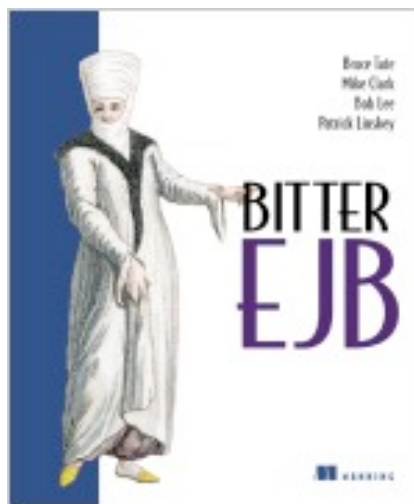
- 🍯 <http://dynaop.dev.java.net/>
- 🍯 <http://aosd.net/>
- 🍯 <http://theserverside.com/>
- 🍯 “Aspect-Oriented Design Pattern Implementations”
 - 🍯 <http://www.cs.ubc.ca/~jan/AODPs/>

Shameless Plugs

🍯 My blog

🍯 <http://java.net/>

🍯 For the more timid...





Thanks for listening!

