TS-5186 - DESIGN PATTERNS RECONSIDERED

Alex Miller http://tech.puredanger.com

Terracotta (Booth 202) http://terracotta.org

Java

Sun microsystems

What is a Design Pattern?

“Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

- Christopher Alexander

“Gang of Four” Design Patterns

Gamma, Helm, Johnson, Vlissides

- Creational
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral
  - Chain of Responsibility
  - Command
  - Interpreter
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

The Patterns Backlash

- Copy/paste
- Design by template
- Cookbook / recipe approach

“The Design Patterns solution is to turn the programmer into a fancy macro processor”

- M. J. Dominus, “Design Patterns” Aren't

The Patterns Backlash

- Design patterns aren't patterns
- Just workarounds for languages missing features

“At code level, most design patterns are code smells.”

- Stuart Halloway

The Patterns Backlash

- Overuse

“Beginning developers never met a pattern or an object they didn't like. Encouraging them to experiment with patterns is like throwing gasoline on a fire.”

- Jeff Atwood, Coding Horror

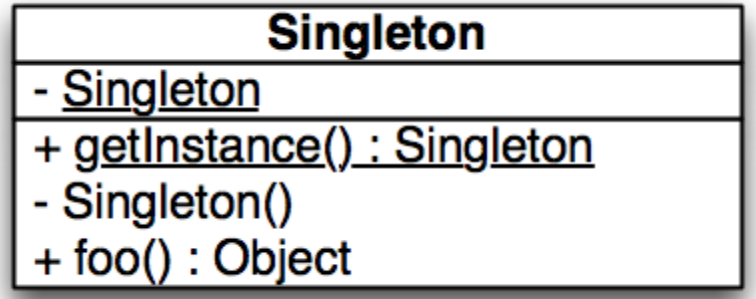
Practical Patterns

- ...form a language or vocabulary
- ...expose real issues
- ...help us compare design choices

Agenda

- Singleton
- Template Method
- Proxy
- Visitor

There can be only one...



Singleton in Code - this is easy!

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

  private Singleton() {
  }

  public static Singleton instance() {
    return INSTANCE;
  }

  public Object read() {
    // nasty database call
    return null;
  }
}
```

Then things go horribly wrong :(

```
public class InnocentBystander {
  public void something() {
    Object foo = Singleton.instance().read();
    // etc...
  }
}
```

```
public class TestInnocentBystander {
  public void testSomething() {
    // Gadzooks! How do I mock the Singleton?
  }
}
```

Singleton has issues

- Hidden coupling
- Testing
- Just one? It's a lie.
- Evolution
- Possible memory leak
- Subclassing
- Initialization order and dependencies

Interfaces to the rescue

```
public interface Singleton {
  Object read();
}


public class SingletonImpl implements Singleton {
  public Object read() {
    // nasty database call
  }
}
```

Dependency injection, you're my hero!

```
public class InnocentBystander {
  private final Singleton singleton;

  public InnocentBystander(Singleton singleton) {
    this.singleton = singleton;
  }

  public void something() {
    Object foo = singleton.read();
    // etc...
  }
}
```

Ah, I feel clean now

```
public class TestInnocentBystander {
  public  void testSomething() {
    Singleton s = new MockSingleton();

    InnocentBystander bystander =
      new InnocentBystander(s);

    bystander.something();

    // assertions
  }
}
```

What have we learned from Singleton?

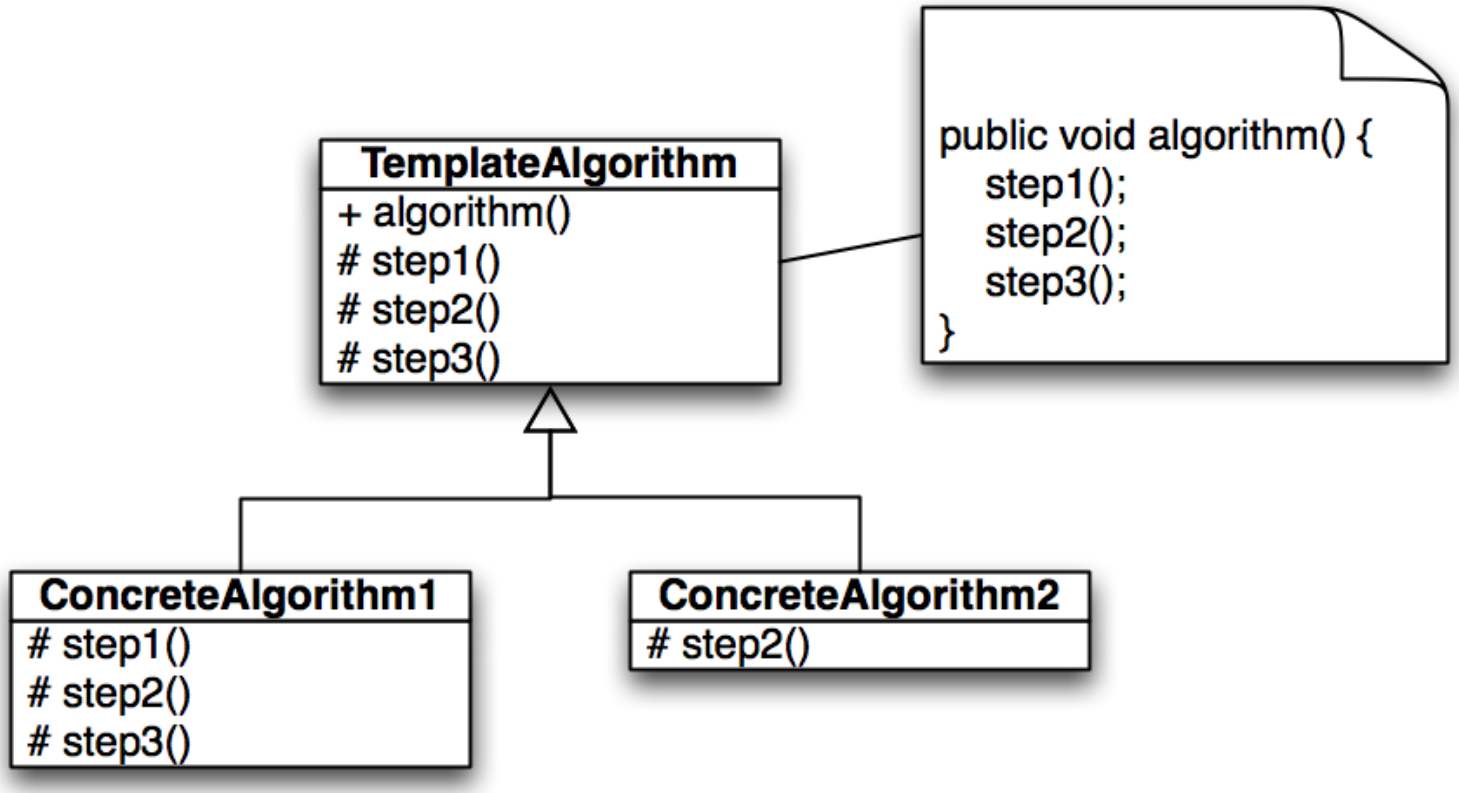
- Interfaces and dependency injection
  - Reduce hidden coupling
  - Allow testability
  - Allow subclassing
  - Make construction and use flexible
- If need only one, control by configuration not by pattern
  - Guice
  - Spring
  - By hand using injection


Singleton

Agenda

- Singleton
- Template Method
- Proxy
- Visitor

Template Method = Pluggable Algorithm



An example in the wild – Spring MVC

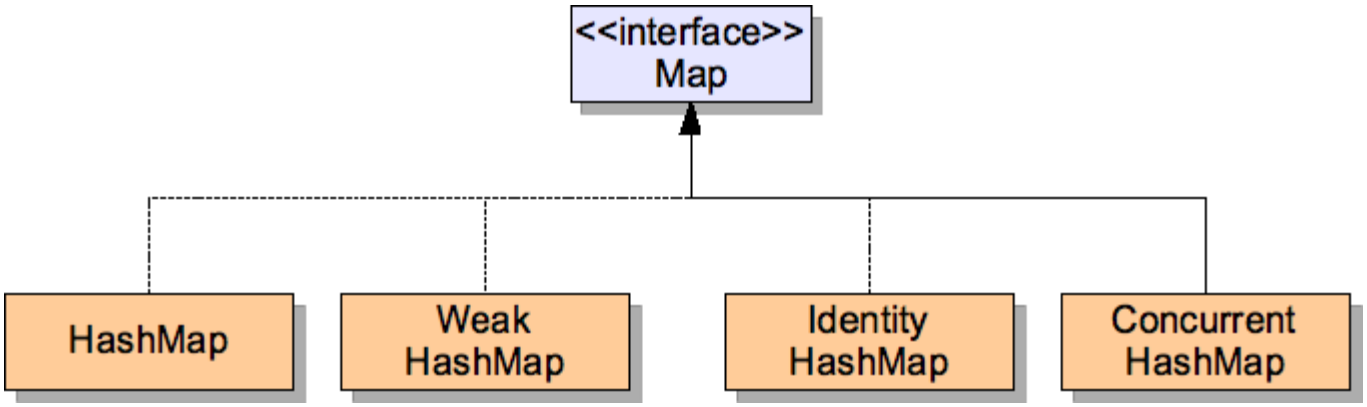
```
Controller (interface)
  AbstractController
    AbstractUrlViewController
      UrlFilenameViewController
    BaseCommandController
      AbstractCommandController
      AbstractFormController
      SimpleFormController
        CancellableFormController
    MultiActionController
    ParamterizableViewController
```

Algorithm spelunking

Workflow (and that defined by superclass):

1. **The controller receives a request for a new form (typically a GET).**
2. Call to `formBackingObject()` which by default, returns an instance of the commandClass that has been configured (see the properties the superclass exposes), but can also be overridden to e.g. retrieve an object from the database (that needs to be modified using the form).
3. Call to `initBinder()` which allows you to register custom editors for certain fields (often properties of non-primitive or non-String types) of the command class. This will render appropriate Strings for those property values, e.g. locale-specific date strings.
4. *Only if `bindOnNewForm` is set to `true`*, then `ServletRequestDataBinder` gets applied to populate the new form object with initial request parameters and the `onBindOnNewForm(HttpServletRequest, Object, BindException)` callback method is called. *Note:* any defined Validators are not applied at this point, to allow partial binding. However be aware that any Binder customizations applied via initBinder() (such as `DataBinder.setRequiredFields(String[])` will still apply. As such, if using bindOnNewForm=true and initBinder() customizations are used to validate fields instead of using Validators, in the case that only some fields will be populated for the new form, there will potentially be some bind errors for missing fields in the errors object. Any view (JSP, etc.) that displays binder errors needs to be intelligent and for this case take into account whether it is displaying the initial form view or subsequent post results, skipping error display for the former.
5. Call to `showForm()` to return a View that should be rendered (typically the view that renders the form). This method has to be implemented in subclasses.
6. The showForm() implementation will call `referenceData()`, which you can implement to provide any relevant reference data you might need when editing a form (e.g. a List of Locale objects you're going to let the user select one from).
7. Model gets exposed and view gets rendered, to let the user fill in the form.
8. **The controller receives a form submission (typically a POST).** To use a different way of detecting a form submission, override the `isFormSubmission` method.
9. If `sessionForm` is not set, `formBackingObject()` is called to retrieve a form object. Otherwise, the controller tries to find the command object which is already bound in the session. If it cannot find the object, it does a call to `handleInvalidSubmit` which - by default - tries to create a new form object and resubmit the form.
10. The `ServletRequestDataBinder` gets applied to populate the form object with current request parameters.
11. Call to `onBind(HttpServletRequest, Object, Errors)` which allows you to do custom processing after binding but before validation (e.g. to manually bind request parameters to bean properties, to be seen by the Validator).
12. If `validateOnBinding` is set, a registered Validator will be invoked. The Validator will check the form object properties, and register corresponding errors via the given `Errors` object.
13. Call to `onBindAndValidate()` which allows you to do custom processing after binding and validation (e.g. to manually bind request parameters, and to validate them outside a Validator).
14. Call `processFormSubmission()` to process the submission, with or without binding errors. This method has to be implemented in subclasses.

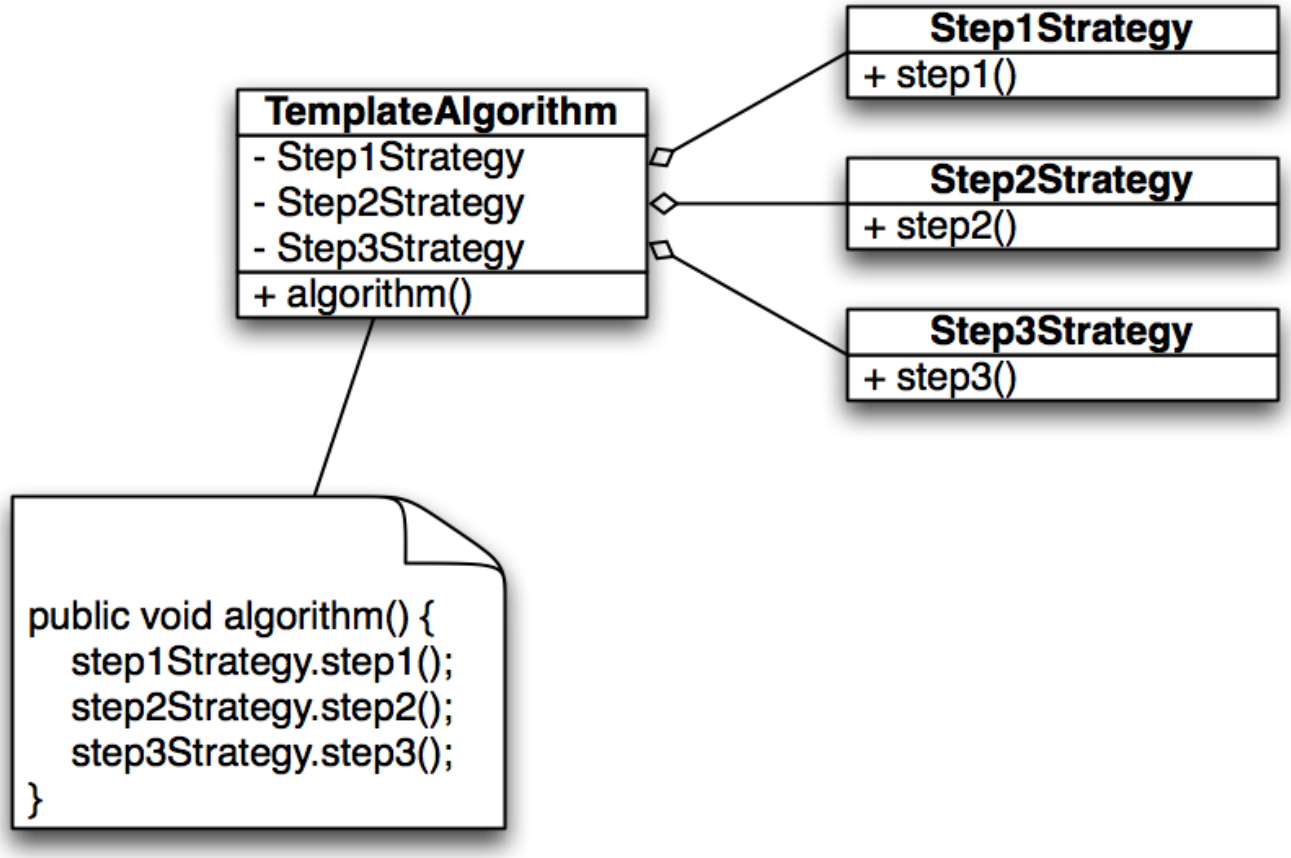
Fighting over your inheritance



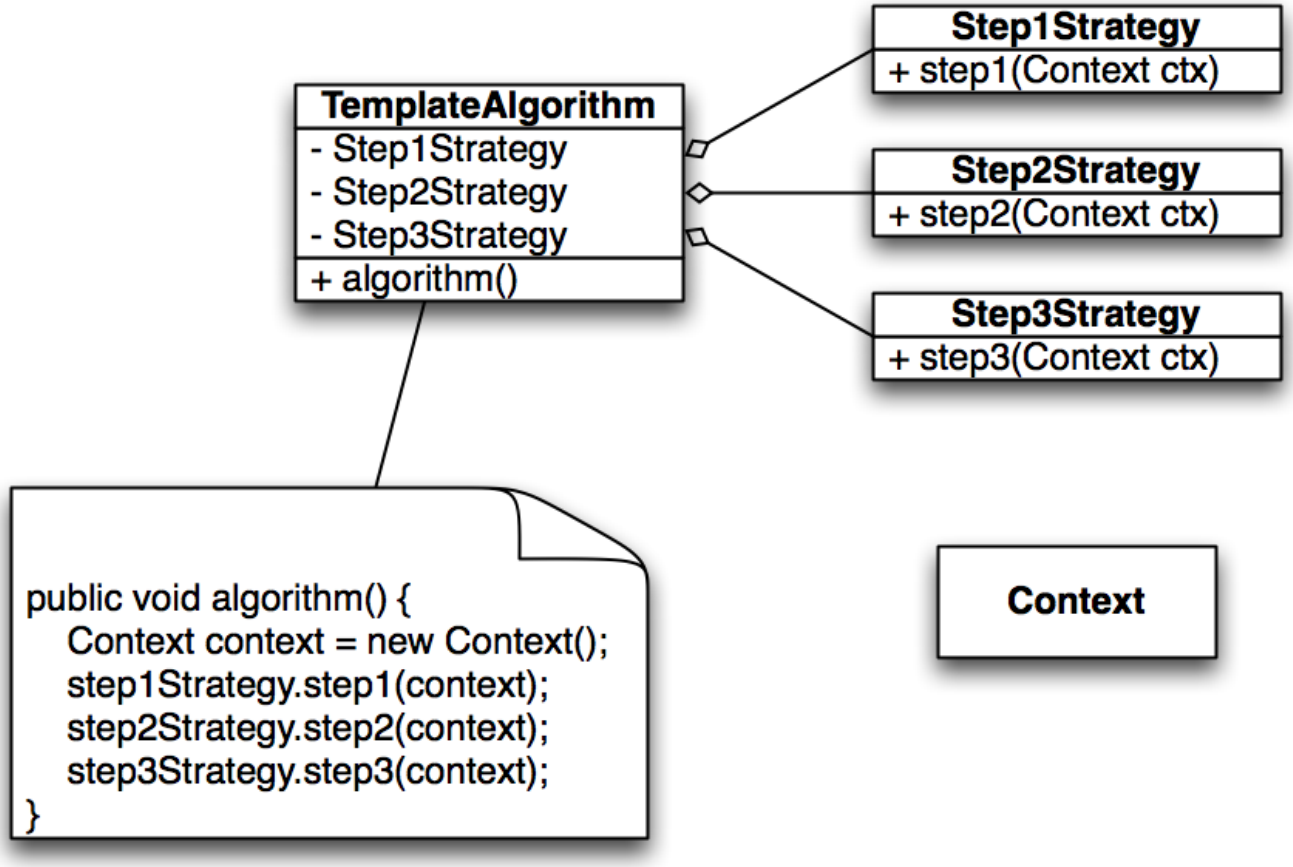
Template Method has issues

- Poorly documents intent to framework user
  - Public vs protected vs abstract methods
  - From different levels of inheritance hierarchy
  - Algorithm and order of calls
- Relies on inheritance
  - Makes composition of functionality difficult
- Hard to maintain and evolve
  - Inheritance hierarchy make completely break down

Replace inheritance with composition



Use context class to expose state



Can closures help?

```
public class TemplateAlgorithm {
  private {Context=>void} step1;

  public void addStep1({Context=>void} block) {
    this.step1=block;
  }

  public  void compute() {
    Context context = new Context();
    step1.invoke(context);
    step2.invoke(context);
    // more steps
  }
}
```

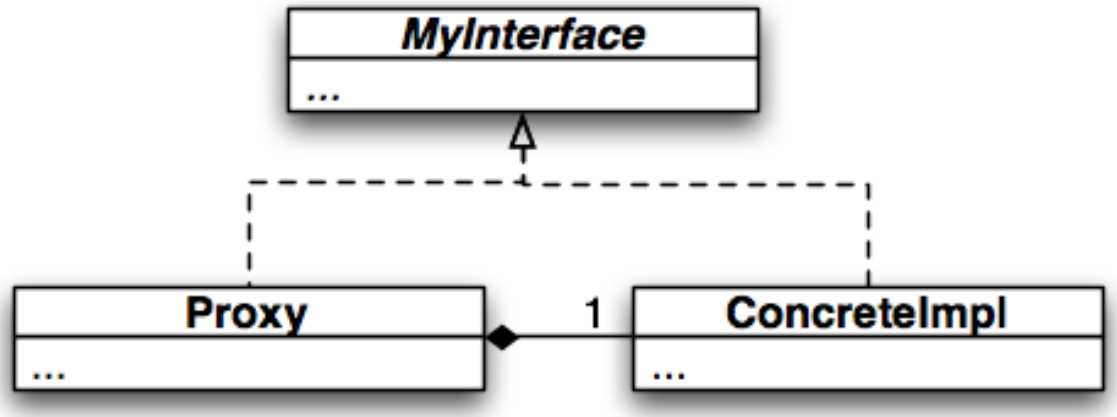
What have we learned from Template Method?

- Prefer composition to inheritance
  - Allows greater reuse
  - Communicates intent better
  - Easier to understand
  - Easier to maintain
  - More robust as it evolves
- Inheritance is a very strong form of coupling
  - Especially in a single-inheritance language

Agenda

- Singleton
- Template Method
- Proxy
- Visitor

Proxy hides instance creation



Have your cake and eat it too!

```
public interface Cake {
  void eat();
}

public class ChocolateCake implements Cake {
  public ChocolateCake() { /* bake */ }
  public void eat()      { /* eat  */ }
}

public class CakeProxy implements Cake {
  private Cake cake;
  public void eat() {
    cake = new ChocolateCake();   // delayed construction
    cake.eat();
  }
}
```

Problem: Tedious to write and maintain static proxy classes

```
public class CakeHandler implements InvocationHandler {
  private Cake cake;

  public Object invoke(Object proxy, Method method,
    Object[] args) throws Throwable {

    if(cake == null) { cake = new ChocolateCake(); }

    if(method.getName().equals("eat")) {
      cake.eat();
      return null;
    } else {
      return method.invoke(method, args);
    }
  }
}
```

Solution #1: Dynamic proxies

Problem: Tedious to write and maintain static proxy classes

Solution #2: AOP

Solution #3: Code generation

Left as an exercise for the reader.... :)

Some other issues...

- Proxy fails when relying on object identity
- Proxy fails when:
  - Subsystems communicate using generic interface
  - Each subsystem downcasts to more specific known type
  - Really a design problem....but it happens
- Dynamic proxy sidesteps static typing by using method names
  - Could use Method.getName() but code is very tedious

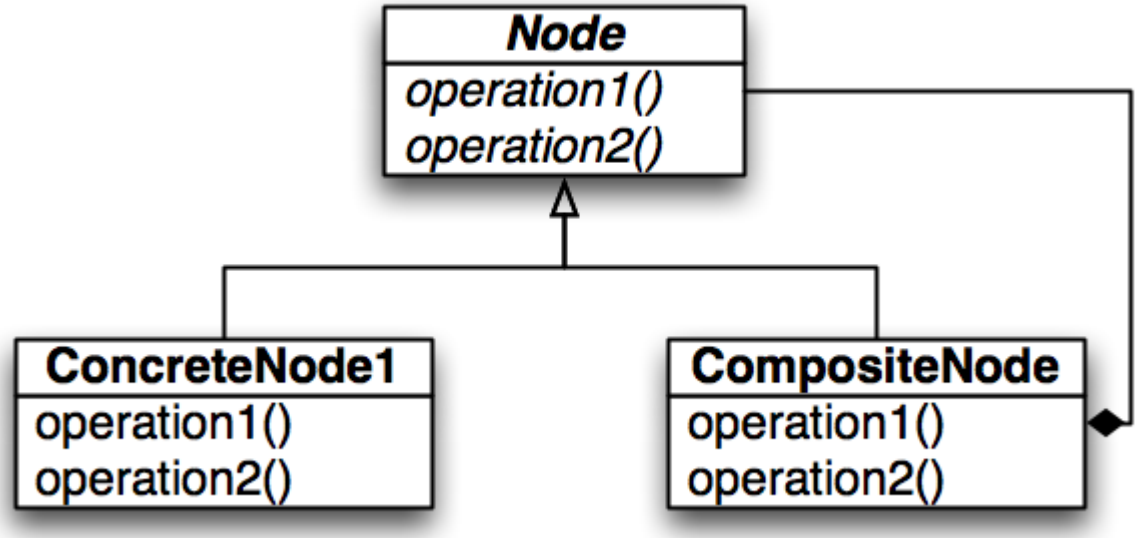
What have we learned from Proxy?

- Proxy implementations have trade-offs
  - Verbosity / conciseness
  - Static typing / type safety
  - Maintenance
  - Performance
- Identity is a tricky issue in OO systems
  - Minimize reliance on identity

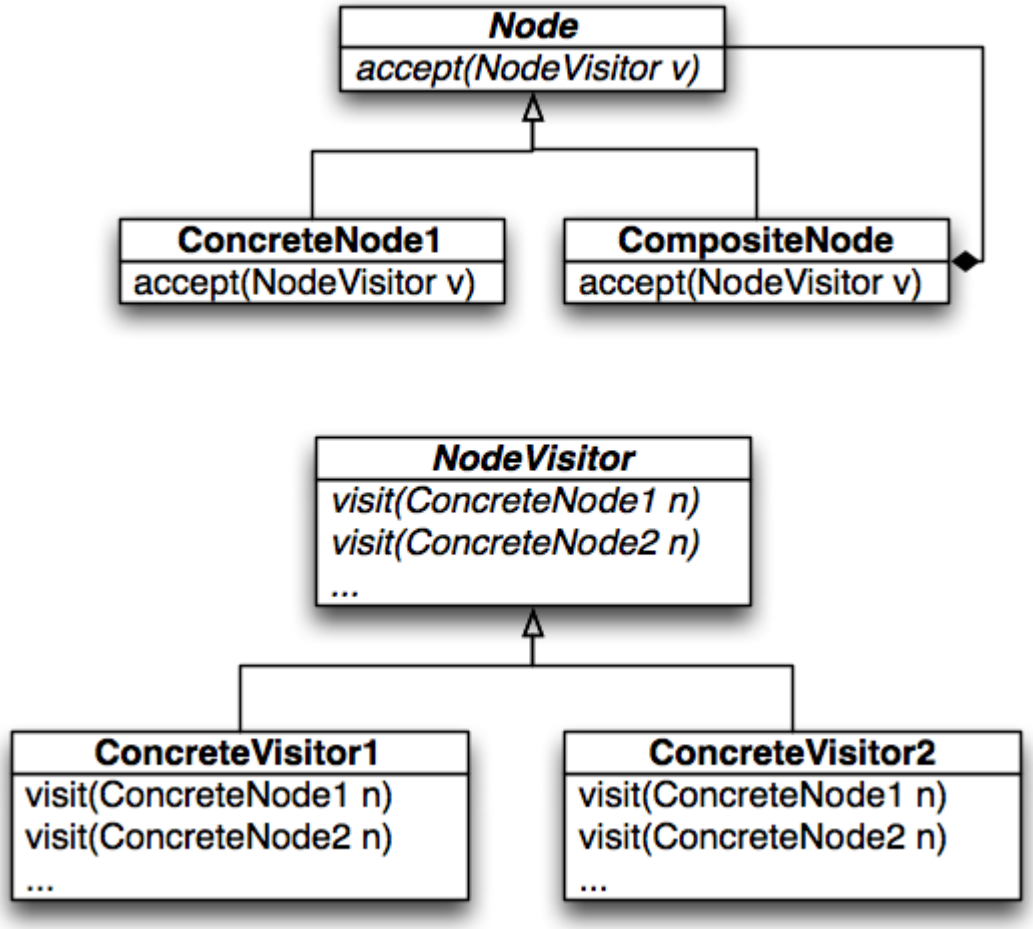
Agenda

- Singleton
- Template Method
- Proxy
- Visitor

Operations over a Composite hierarchy



Stopping by for a visit



Implementing Visitor

```
public interface Visitable {
  void acceptVisitor(Visitor visitor);
}

public interface Visitor {
  void visit(ConcreteNode1 node1);
  // repeat for all concrete nodes
}

public class ConcreteNode1 implements Visitable {
  public void acceptVisitor(Visitor visitor) {
    visitor.visit(this);
  }
}

public class ConcreteVisitor implements Visitor {
  // ...
}
```

The Expression Problem - Philip Wadler, 1998

- Add new cases to a data type
- Add new functions over the data type
- Don't recompile when adding either cases or functions
- Don't lose static type safety

Problem: Where does navigation code live?

```
public class CompositeNode implements Visitable {
  private List<Node> nodes;

  public void acceptVisitor(Visitor visitor) {
    visitor.visit(this);
    for(Node node : nodes) {
      node.acceptVisitor(visitor);
    }
  }
}
```

Solution #1: In nodes

Problem: Where does navigation code live?

```
public class NavigationVisitor extends BaseVisitor {
  private Visitor logicVisitor;
  private LinkedList<? super Visitable> nodeQueue =
    new LinkedList<Visitable>();

  public NavigationVisitor(Visitor logicVisitor) {
    this.logicVisitor = logicVisitor);
  }

  private void visitNext() {
    if(!nodeQueue.isEmpty()) {
      Visitable next = (Visitable)nodeQueue.removeFirst();
      next.acceptVisitor(this);
    }
  }
```

Solution #2: In navigation visitor

Problem: Where does navigation code live?

```
  public visit(ConcreteNode1 node) {
    logicVisitor.visit(node);
    visitNext();
  }

  public visit(CompositeNode node) {
    nodeQueue.addAll(node.getChildren());
    visitNext();
  }

  // etc
}
```

Solution #2: In navigation visitor

Common Visitor Types

- "Collector" visitor
  - Collect and accumulate for return
- "Finder" visitor
  - Return immediately when match found
- "Event" visitor
  - Stateless, fire events for subset of nodes
- "Transform" visitor
  - Modify the tree while walking it
  - Potentially dangerous :)
- "Validation" visitor
  - Verifies the structure and reports problems

Problem: Returning a Value

```
public class CollectorVisitor extends BaseVisitor {
  private final List<Foo> collected =
    new ArrayList<Foo>();

  public List<Foo> getCollected() {
    return this.collected;
  }

  public void visit(ConcreteNode1 node) {
    if(shouldCollect(node)) {
      collected.add(node);
    }
  }
}
```

Solution #1: Collect in visitor

Problem: Returning a Value

```
public interface Visitable {
  <C> void acceptVisitor(Visitor<C> visitor, C context);
}

public interface Visitor<C> {
  void visit(ConcreteNode1 node1, C context);
  // for all concrete nodes
}

public class ConcreteNode1 implements Visitable {
  public <C> void acceptVisitor(Visitor<C> visitor,
                                C context) {
    visitor.visit(this, context);
  }
}
```

Solution #2: Generics

Problem: Returning a Value

```
public class ConcreteVisitor implements
Visitor<List<String>> {

  public void visit(ConcreteNode1 node,
                    List<String> context) {

    context.add(node.getStr());
  }
}

// example use
ConcreteNode root = ...
ConcreteVisitor v = new ConcreteVisitor();
List<String> context = new ArrayList<String>();
root.acceptVisitor(v, context);
```

Solution #2: Generics

Problem: Exceptions

```
public class ExceptionalVisitor extends BaseVisitor {
  private MyException exception;

  private ExceptionalVisitor() {}

  public void checkError() throws MyException {
    if(this.exception != null) {
      throw exception;
    }
  }

  public static void run(Node root) throws MyException {
    ExceptionalVisitor visitor = new ExceptionalVisitor();
    root.acceptVisitor(visitor);
    checkError();
  }
}
```

Solution #1: Store in ConcreteVisitor

Problem: Exceptions

```
public interface Visitable {
  <E extends Exception>
  void acceptVisitor(Visitor<E> visitor) throws E;
}

public interface Visitor<E extends Exception> {
  void visit(ConcreteNode1 node1) throws E;
  // for all concrete nodes
}

public class ConcreteNode1 implements Visitable {
  <E extends Exception>
  public void acceptVisitor(Visitor<E> visitor) throws E {
    visitor.visit(this);
  }
}
```

Solution #2: Generics

Can closures help?

- What if we define the “visit” for each type as a closure?

- Collector visitor – can collect into local state of caller
- Finder visitor – use non-local return to abort & return
- Validation visitor – use non-local return to abort & return

- Simplify visitors with many similar methods
  - Dynamically assemble with closures

DynamicVisitor built from closures

```
public class DynamicVisitor implements Visitor {
  private {ConcreteNode1==>void} concreteNode1Block;
  private {ConcreteNode2==>void} concreteNode2Block;
  private {CompositeNode==>void} compositeNodeBlock;

  public void addConcreteNode1(
    {ConcreteNode1==>void} block) {
    this.concreteNode1Block = block;
  }

  public void visit(ConcreteNode1 node) {
    if(concreteNode1Block != null) {
      concreteNode1Block.invoke(node);
    }
  }

  ...
}
```

VisitorBuilder

```
public class VisitorBuilder {
  public static VisitorBuilder visitor() {
    return new VisitorBuilder();
  }

  private DynamicVisitor visitor = new DynamicVisitor();

  public VisitorBuilder handleConcreteNode1(
    {ConcreteNode1==>void} block) {

    visitor.addConcreteNode1(block);
    return this;
  }

  public Visitor build() {
    return this.visitor;
  }
}
```

Assemble

```
public class TestFinder {
  public static Node findMatch(String name, Node root) {
    Visitor visitor = VisitorBuilder.visitor()
      .handleConcreteNode1({ConcreteNode1 node ==>
          if(node.getName().equals(name)) {
            return node;
          }
        })
      .handleConcreteNode2({ConcreteNode2 node ==>
          if(node.getName().equals(name)) {
            return node;
          }
        })
      .build();
    root.accept(visitor);
    return null;
  }
}
```

What have we learned from Visitor?

- Expression problem is tough to crack
- Separate parts of the design that change at different rates
- Generics add precision and flexibility by exposing hidden coupling
- Closures provide some interesting new possibilities

Design Principles

- Use interfaces and dependency injection to reduce coupling
- Favor composition over inheritance
- Separate logic that will evolve at different rates
- Rely on object identity as little as possible
- Leverage static typing and generics

Summary

Design patterns are a valuable tool to describe real programming problems.

Solutions to all design problems are contextual (dependent on language, the code base, and the developers themselves).

Use design patterns as a starting to point to discuss alternatives. Don't stop at an example from a book or a blog or even a JavaOne℠ event presentation. :)

Experiment to find what will work best for you!

THANK YOU

Design Patterns Reconsidered
Alex Miller

http://tech.puredanger.com/presentations