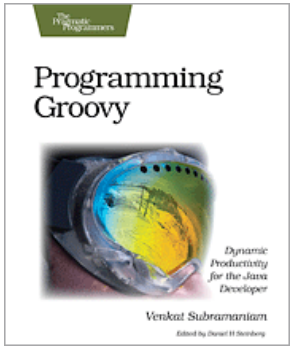
DESIGN PATTERNS IN JAVA AND GROOVY

```
speaker.identity {
  name        'Venkat Subramaniam'
  company     'Agile Developer, Inc.'
  credentials 'Programmer', 'Author', 'Trainer'
  blog        'http://agiledeveloper.com/blog'
  email       'venkats@agiledeveloper.com'
}
```



Abstract

- You're most likely familiar with the Gang-of-four design patterns and how to implement them in Java. However, you wouldn't want to implement those patterns in a similar way in Groovy. Furthermore, there are a number of other useful patterns that you can apply in Java and Groovy. In this presentation we'll look at two things: How to use patterns in Groovy and beyond Gang-of-four patterns in Groovy and Java.
- Patterns overview
- Implementing common patterns in Groovy
- Beyond Gang-of-four patterns in Java and Groovy
- Lots of examples

Abstract Factory

- You want to abstract the object creation process
- Extensible to add new types of objects
- Helpful to create consistent families of objects

Abstract Factory

- I have a set of operation that I want to perform on objects of different type. I need to create these objects first, then call methods.
- In Java, the `Class` class acts as a factory as well

Abstract Factory–Java

```
public static Object create(String klass, HashMap properties) throws Exception
{
    Object instance = Class.forName(klass).newInstance();
    for(Object property : properties.keySet())
    {
      PropertyUtils.setSimpleProperty(instance,
          property.toString(), properties.get(property));
    }

    return instance;
}
```

Abstract Factory–Groovy

- In Groovy, name of class refers to `Class` object

```
println Book    //=> class Book
println Journal //=> clas Journal
```

- Groovy GDK also has added methods to `Class`

```
def create(kclass, properties)
{
  def instance = kclass.newInstance()
  properties.each { name, value ->
    instance."${name}" = value
    }

  instance
}

println create(Book, [title: 'Who moved my Cheese', pages: 96])
println create(Journal, [title: 'ACM Communications', volume: 2])
```

```
Who moved my Cheese:96
ACM Communications:2
```

Pluggable Behavior-Strategy

- You want to vary the guts or internal implementation of certain algorithm or task
- You want to be able to plug some behavior or variations in the middle of a certain computation or task

Pluggable Behavior–Java

```
public static int totalSelectValues(int number, Criteria criteria)
{
  int sum = 0;
  for(int i = 1; i <= number; i++)
  {
    if (criteria.evaluate(i)) sum += i;
  }

  return sum;
}

public static void main(String[] args)
{
  int result = 0;

  result = totalSelectValues(10, new Criteria() {
    public boolean evaluate(Object object)
    {
      return (Integer)(object) % 2 == 0;
    }
  });

  System.out.println("Total of even numbers from 1 to 10 is " + result);
}
```

Pluggable Behavior–Groovy

- Closures in Groovy make it real simple to realize this

```
def totalSelectValues(number, closure)
{
  def sum = 0
  1.upto(number) {
    if (closure(it)) sum += it
  }

  sum
}

result = totalSelectValues(10) { it %2 == 0 }

println "Total of even numbers from 1 to 10 is ${result}"

Total of even numbers from 1 to 10 is 30
```

Pluggable Behavior–Groovy

- You can store away the pluggable code for later use, if you like

```
class Equipment
{
  final calc
  Equipment(calculator) { calc = calculator }

  def compute()
  {
    print "Computing using "
    calc()
  }
}
```

```
equipment1 = new Equipment() { println "a calculator" }

anotherCalculator = { println "another calculator" }
equipment2 = new Equipment(anotherCalculator)
equipment3 = new Equipment(anotherCalculator)

equipment1.compute()
equipment2.compute()
equipment3.compute()
```

```
Computing using a calculator
Computing using another calculator
Computing using another calculator
```

Execute Around Method(EAM)

- You have a pair of operation that needs to be performed before and after operations
- You have a resource that needs to be opened/connected to and then safely/automatically closed
- You want deterministic control of when resource is deallocated (after all you can't rely on the finalizer)

EAM–Java

```
class Resource
{
  private void open() { System.out.println("opened..."); }
  private void close() { System.out.println("closed..."); }

  public void operation1() { System.out.println("operation1..."); }
  public void operation2() { System.out.println("operation2..."); }

  public static void create(UseResource useResource)
  {
    Resource resource = new Resource();
    try
    {
      resource.open();
      useResource.use(resource);
    }
    finally
    {
      resource.close();
    }
  }
```

```
interface UseResource
{
  public void use(Resource r);
}

public static void main(String[] args)
{
  Resource.create(new UseResource() {
    public void use(Resource r)
    {
      r.operation1();
      r.operation2();
    }
  });
}
```

Hmm, not very elegant, is it?

EAM–Groovy

```groovy
class Resource
{
  void open() { println('opened...') }
  void close() { println('closed...') }

  void operation1() { println("operation1...") }
  void operation2() { println("operation2...") }

  static void create(closure)
  {
    def resource = new Resource()
    try
    {
      resource.open()
      closure(resource)
    }
    finally
    {
      resource.close()
    }
  }
}
```

```groovy
Resource.create() { r ->
  r.operation1()
  r.operation2()
}
```

```
opened...
operation1...
operation2...
closed...
```

Iterator

- You want to traverse a collection of objects in a consistent manner independent of the type of collection
- An external iterator allows you to control the actual traversal
- An internal iterator takes care of that—you provide code that needs to be executed for each element in the collection

Iterator–Java

- You can use `for` (for-each) on any class that implements `Iterable` interface—external iterator

```
public static void main(String[] args)
{
  Car car = new Car();

  for(Wheel wheel : car)
  {
    System.out.println(wheel);
  }
}
```

```
class Wheel {}
class Car implements Iterable<Wheel>
{
  private Wheel[] wheels = new Wheel[4];

  public Car()
  {
    for(int i = 0; i < 4; i++) { wheels[i] = new Wheel(); }
  }

  public java.util.Iterator<Wheel> iterator()
  {
    return new java.util.Iterator<Wheel>()
    {
      int index = 0;
      public boolean hasNext() { return index < 4; }
      public Wheel next() { return Car.this.wheels[index++]; }
      public void remove() { throw new UnsupportedOperationException(); }
    };
  }
```

Iterator–Groovy

- Groovy `each` method provides internal iterator

```
class Wheel {}
class Car
{
  final wheels = []

  Car() { 4.times { wheels << new Wheel() } }

  def eachWheel(closure)
  {
    wheels.each { closure(it) }
  }
}

Car car = new Car();
car.eachWheel { wheel -> println wheel }
```

```
Wheel@10c68663
Wheel@23a53800
Wheel@5d1859fe
Wheel@4cbc2950
```

Cascade

- You want to perform a series of operations on an object
- The operations form a cohesive sequence or thread of calls
- Provides a context, reduces noise
- Call the method on the result of previous call or establish a context object

Cascade–Java

```
class Mailer
{
  public Mailer to(String address)
  {// ...
    return this;
  }

  public Mailer from(String address)
  {// ...

  public Mailer cc(String address)
  {// ...

  //...

  public Mailer subject(String sub)
  {// ...

  public void send(String message)
  {

  public static void main(String[] args)
  {
    new Mailer().from("venkats@agiledeveloper.com")
      .to("test@agiledeveloper.com")
      .subject("test")
      .send("test message");
  }
}
```

Cascade–Groovy

- Groovy provides a special set of methods: `with` and `identity`

```
class Mailer
{
  void to(String address)
  {
    // ...
  }

  void from(String address)
  {...

  void cc(String address)
  {...

  //...

  void subject(String subj)
  {...

  void send(String message)
  {...
}
```

```
new Mailer().with {
  from('venkats@agiledeveloper.com)
  to('test@agiledeveloper.com')
  subject('test')
  send('test message')
}
```

Intercept, Cache, Invoke

- You're interested in synthesizing method
- However, you don't want to take a performance hit each time your dynamic method is called
- You want to intercept the call, create and cache the implementation, and invoke it

Intercept, Cache, Invoke

```groovy
class Person
{
  def work() { "working..." }

  def plays = ['Tennis', 'VolleyBall', 'BasketBall']

  def methodMissing(String name, args)
  {
    System.out.println "methodMissing called for $name"
    def methodInList = plays.find { it == name.split('play')[1]}

    if (methodInList)
    {
      def impl = { Object[] vargs ->
          return "playing $name..."
        }

      Person.metaClass."$name" = impl //future calls will use this

      return impl(args)
    }
    else
    {
      throw new MissingMethodException(name, Person.class, args)
    }
  }

  static { Person.metaClass }
}
```

```groovy
jack = new Person()
println jack.work()
println jack.playTennis()
println jack.playTennis()
```

```
working...
methodMissing called for playTennis
playing playTennis...
playing playTennis...
```

Delegation

- You want to delegate calls to methods of another object
- You prefer to use delegation over inheritance

Delegation–Groovy

```groovy
class Worker
{
  def simpleWork1(spec) { println "worker does work1 with spec $spec" }
  def simpleWork2() { println "worker does work2" }
}

class Expert
{
  def advancedWork1(spec) { println "Expert does work1 with spec $spec" }
  def advancedWork2(scope, spec)
  {
    println "Expert does work2 with scope $scope spec $spec"
  }
}
```

```groovy
class Manager
{
  def worker = new Worker()
  def expert = new Expert()

  def schedule() { println "Scheduling ..." }

  def methodMissing(String name, args)
  {...}
}
```

See Next Page

Delegation–Groovy...

```groovy
def methodMissing(String name, args)
{
  println "intercepting call to $name..."
  def delegateTo = null

  if(name.startsWith('simple')) { delegateTo = worker }
  if(name.startsWith('advanced')) { delegateTo = expert }

  if (delegateTo?.metaClass.respondsTo(delegateTo, name, args))
  {
    Manager.metaClass."${name}" = { Object[] varArgs ->
          return delegateTo.invokeMethod(name, *varArgs)
    }

    return delegateTo.invokeMethod(name, args)
  }

  throw new MissingMethodException(name, Manager.class, args)
}
```

Delegation–Groovy...

```
peter = new Manager()
peter.schedule()
peter.simpleWork1('fast')
peter.simpleWork1('quality')
peter.simpleWork2()
peter.simpleWork2()
peter.advancedWork1('fast')
peter.advancedWork1('quality')
peter.advancedWork2('protype', 'fast')
peter.advancedWork2('product', 'quality')
try
{
  peter.simpleWork3()
}
catch(Exception ex)
{
  println ex.message
}
```

```
Scheduling ...
intercepting call to simpleWork1...
worker does work1 with spec fast
worker does work1 with spec quality
intercepting call to simpleWork2...
worker does work2
worker does work2
intercepting call to advancedWork1...
Expert does work1 with spec fast
Expert does work1 with spec quality
intercepting call to advancedWork2...
Expert does work2 with scope protype spec fast
Expert does work2 with scope product spec quality
intercepting call to simpleWork3...
No signature of method: Manager.simpleWork3() is applicable f
```

Delegation–Groovy...

- That's nice, but...
- That's a lot of work
- You don't want to pay that toll each time you want to delegate

- Let's refactor

Delegation–Refactored

Elegant, eh?!

```
class Manager
{
  { delegateCallsTo Worker, Expert, GregorianCalendar }

  def schedule() { println "Scheduling ..." }
}
```

What's that?

Next Page...

Delegation–Refactored

```
Object.metaClass.delegateCallsTo = {Class... klassOfDelegates ->

  def objectOfDelegates = klassOfDelegates.collect { it.newInstance() }

  delegate.metaClass.methodMissing = { String name, args ->
      println "intercepting call to $name..."

      def delegateTo = objectOfDelegates.find {
                    it.metaClass.respondsTo(it, name, args) }

      if (delegateTo)
      {
        delegate.metaClass."${name}" = { Object[] varArgs ->
            def params = varArgs?:null
            return delegateTo.invokeMethod(name, *params)
        }

        return delegateTo.invokeMethod(name, args)
      }
      else
      {
        throw new MissingMethodException(name, delegate.getClass(), args)
      }
    }
}
```

References

- Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, Addison-Wesley.
- Smalltalk Best Practice Patterns, by Kent Beck, Prentice Hall.
- Programming Groovy: Dynamic Productivity for the Java Developers—Venkat Subramaniam, Pragmatic Bookshelf.

You can download examples and slides from

http://www.agiledeveloper.com - download

Thank You!

Please fill in your session evaluations