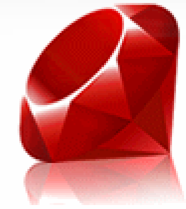
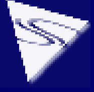


Submission 642 © ASERT 2007

Agile Developer Practices for Dynamic Languages

Dr Paul King & Rhys Keepence

ASERT, Australia



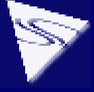
Introduction …

- **Developer practices**
  - **Well understood and documented for traditional and agile approaches such as Java development**
  - **But dynamic languages like Groovy and Ruby change the ground rules**
  - **Many of the rules and patterns we have been taught no longer apply**
- **This tutorial explores some of these changes**

```
cities  = %w[ London
              Oslo
              Paris
              Amsterdam
              Berlin ]
visited = %w[Berlin Oslo]

puts "I still need " +
     "to visit the " +
     "following cities:",
     cities - visited
```

```
cap = 'o' * 180
while (m = (cap =~ /^(oo+?)\1+$/)) {
    p1 = m[0][1]
    print p1.size() + ' '
    cap = cap.replaceAll(p1,'o')
}
println cap.size()
// => 2 2 3 3 5
```

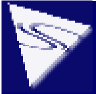
Submission 642 © ASERT 2007



… Introduction

- **Traditional Developer Practice Guidelines**
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995). ***Design Patterns**: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
  - Martin Fowler (1999). ***Refactoring**: Improving the Design of Existing Code*. Addison-Wesley.
  - Joshua Bloch (2001). ***Effective Java Programming** Language Guide*. Prentice Hall.
  - Robert Martin (2002), ***Agile Software Development**, Principles, Patterns, and Practices*. Prentice Hall.
- **Where are they now?**

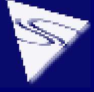
Submission 642 © ASERT 2007



Static vs Dynamic Typing …

- **Static Typing Pros**
  - Errors are often detected earlier and with better error messages
  - Code can sometimes be clearer – you don't need to infer the types to understand the code – especially when revisiting the code later
  - Code can be more declarative – otherwise we might need procedural tests
  - Better refactoring, editing and other forms of source processing support is often possible
  - Better optimisations are often possible
  - Often easier to understand a system from the outside (statically-typed APIs and interfaces)
  - With generics support you can start to nail down even complex cases

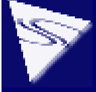
Submission 642 © ASERT 2007



… Static vs Dynamic Typing …

- **Dynamic Typing Pros**
  - Speed development through duck-typing and less boiler-plate code
  - Allow more expressiveness
  - Easy to learn
  - You need to cater for things static typing misses anyway, why not cover everything
  - You need to have tests for other parts of your functionality, why not cover off types as part of your other tests
  - No false sense of security
  - Less likely to neglect good documentation and/or good coding conventions on the grounds that your static types make everything clear by themselves

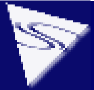
Submission 642 © ASERT 2007



… Static vs Dynamic Typing

- **Guidelines Groovy/Ruby**
  - Don't be afraid of throwing exceptions or handling unexpected messages using metaprogramming
- **Guidelines Groovy**
  - If you are going to use Interfaces, use fine-grained interfaces (Interface Oriented Design) or throw exceptions
  - Use static types whenever you feel it aids with clarity of the program or increases type safety, e.g. financial calculations
  - Static typing promotes stability, dynamic typing promotes flexibility

Submission 642 © ASERT 2007



Language features instead of Patterns …

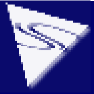
```
class SquarePeg {
    def width
}

class RoundPeg {
    def radius
}

class RoundHole {
    def radius
    def pegFits(peg) {
        peg.radius <= radius
    }
    String toString() { "RoundHole with radius $radius" }
}

class SquarePegAdapter {
    def peg
    def getRadius() {
        Math.sqrt(((peg.width/2) ** 2)*2)
    }
    String toString() {
        "SquarePegAdapter with peg width $peg.width (and notional radius $radius)"
    }
}
```

```
def hole = new RoundHole(radius:4.0)
(4..7).each { w ->
    def peg = new SquarePegAdapter(peg:new SquarePeg(width:w))
    if (hole.pegFits(peg))
        println "peg $peg fits in hole $hole"
    else
        println "peg $peg does not fit in hole $hole"
}
```

```
def peg = new SquarePeg(width:w)
peg.metaClass.radius = Math.sqrt(((peg.width/2) ** 2)*2)
```

Adapter Pattern
Do I create a whole new class or just add the method I need on the fly?

Submission 642 © ASERT 2007



… Language features instead of Patterns …

```groovy
package plain

abstract class Shape { }

class Rectangle extends Shape {
    def x, y, width, height

    Rectangle(x, y, width, height) {
        this.x = x; this.y = y; this.width = width; this.height = height
    }

    def union(rect) {
        if (!rect) return this
        def minx = [rect.x, x].min()
        def maxx = [rect.x + width, x + width].max()
        def miny = [rect.y, y].min()
        def maxy = [rect.y + height, y + height].max()
        new Rectangle(minx, miny, maxx - minx, maxy - miny)
    }

    def accept(visitor) {
    visitor.visit_rectangle(this)
}
}

class Line extends Shape {
    def x1, y1, x2, y2

    Line(x1, y1, x2, y2) {
        this.x1 = x1; this.y1 = y1; this.x2 = x2; this.y2 = y
    }

    def accept(visitor){
    visitor.visit_line(this)
    }
}

class Group extends Shape {
    def shapes = []
    def add(shape) { shapes += shape }
    def remove(shape) { shapes -= shape }
    def accept(visitor) {
    visitor.visit_group(this)
}
}

class BoundingRectangleVisitor {
    def bounds

    def visit_rectangle(rectangle) {
        if (bounds)
            bounds = bounds.union(rectangle)
        else
            bounds = rectangle
    }

    def visit_line(line) {
        def line_bounds = new Rectangle(line.x1, line.y1, line.x2-line.y1, line.x2-line.y2)
        if (bounds)
            bounds = bounds.union(line_bounds)
        else
            bounds = line_bounds
    }

    def visit_group(group) {
        group.shapes.each { shape -> shape.accept(this) }
    }
}

def group = new Group()
group.add(new Rectangle(100, 40, 10, 5))
group.add(new Rectangle(100, 70, 10, 5))
group.add(new Line(90, 30, 60, 5))
def visitor = new BoundingRectangleVisitor()
group.accept(visitor)
bounding_box = visitor.bounds
println bounding_box.dump()
```

Visitor Pattern

<= without closures

with closures =>

```groovy
abstract class Shape {
    def accept(Closure yield) { yield(this) }
}

class Rectangle extends Shape {
    def x, y, w, h
    def bounds() { this }
    def union(rect) {
        if (!rect) return this
        def minx = [rect.x, x].min()
        def maxx = [rect.x + w, x + w].max()
        def miny = [rect.y, y].min()
        def maxy = [rect.y + h, y + h].max()
        new Rectangle(x:minx, y:miny, w:maxx - minx, h:maxy - miny)
    }
}

class Line extends Shape {
    def x1, y1, x2, y2
    def bounds() {
        new Rectangle(x:x1, y:y1, w:x2-y1, h:x2-y2)
    }
}

class Group {
    def shapes = []
    def leftShift(shape) { shapes += shape }
    def accept(Closure yield) { shapes.each{it.accept(yield)} }
}

def group = new Group()
group << new Rectangle(x:100, y:40, w:10, h:5)
group << new Rectangle(x:100, y:70, w:10, h:5)
group << new Line(x1:90, y1:30, x2:60, y2:5)
def bounds
group.accept{ bounds = it.bounds().union(bounds) }
println bounds.dump()
```

Submission 642 © ASERT 2007

… Language features instead of Patterns

```
interface Calc {
    def execute(n, m)
}

class CalcByMult implements Calc {
    def execute(n, m) { n * m }
}

class CalcByManyAdds implements Calc {
    def execute(n, m) {
        def result = 0
        n.times{
            result += m
        }
        return result
    }
}

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

Calc[] multiplicationStrategies = [
    new CalcByMult(),
    new CalcByManyAdds()
]

sampleData.each{ data ->
    multiplicationStrategies.each{ calc ->
        assert data[2] == calc.execute(data[0], data[1])
    }
}
```

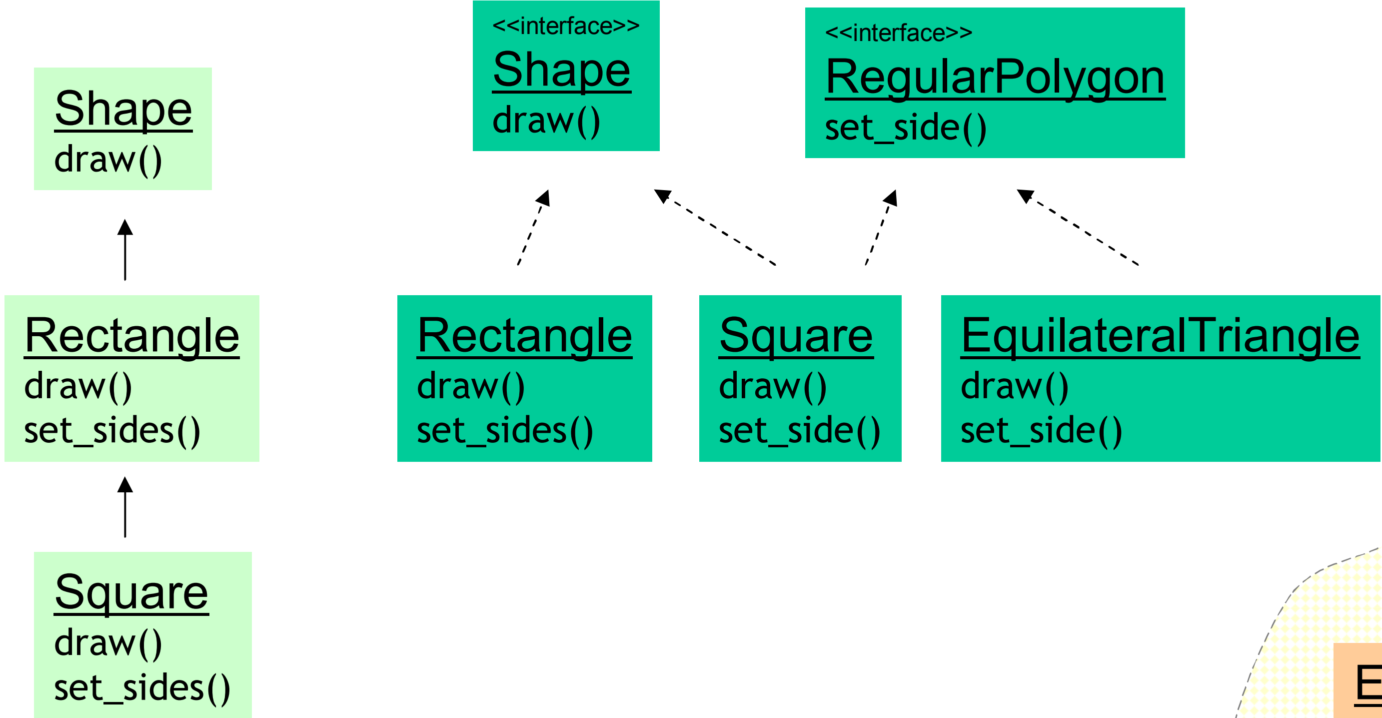
Strategy Pattern
<= with interfaces
with closures =v

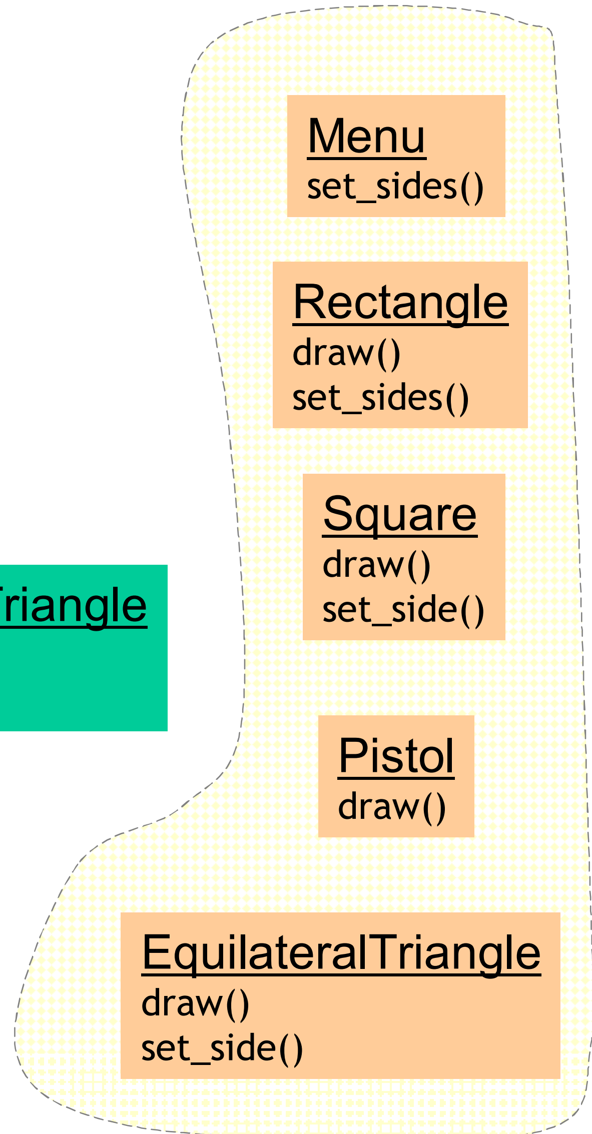
```
def multiplicationStrategies = [
    { n, m -> n * m },
    { n, m -> def result = 0; n.times{ result += m }; result }
]

def sampleData = [
    [3, 4, 12],
    [5, -5, -25]
]

sampleData.each{ data ->
    multiplicationStrategies.each{ calc ->
        assert data[2] == calc(data[0], data[1])
    }
}
```

Submission 642 © ASERT 2007

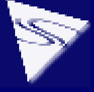
Interface-Oriented Design vs Duck Typing

Adapted from [2]





Submission 642 © ASERT 2007



Flexible Mocks …

- **Mocking is an important aspect to interaction-based testing. You don't always need a complex mocking package:**
  - **Stubs**
  - **Closures**
  - **Maps**
  - **Metaprogramming**

Submission 642 © ASERT 2007

… Flexible Mocks

Traditionally, we'd use mocks for factory and logger, here we use Expandos and Maps =>

```
class MyApp {
    def factory
    def logger
    def doBusinessLogic(param) {
        def myObj = factory.instance
        myObj.doSomething(param)
        myObj.doSomethingElse(param)
        logger.log('Something done with: ' + param)
    }
}

def factory = [instance: businessObj]

def logger = new Expando()
logger.log = { msg -> assert msg == 'Something done with: ' + param }
```

```
class ClosureMocking {
    String methodOne() {
        methodTwo()
    }
    String methodTwo() {
        "television"
    }
}

def emc = new ExpandoMetaClass(ClosureMocking,true)
emc.methodTwo = {-> "radio" }
emc.initialize()

assert "radio" == new ClosureMocking().methodOne()
```

<= 'mock' using metaprogramming

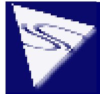
Submission 642 © ASERT 2007

DSLs …

- **Creating a Testing DSL**
  - **Ideally, acceptance tests should be written by your customer**
  - **Ruby is a malleable language, conducive to creating natural, understandable code**
  - **Example:**

```
add_book_with :name => "Groovy In Action",
              :author => "Paul King",
              :cost => 10.dollars
create_new_user_with :email => "jim@jim.com",
                     :name => "Jim",
                     :password => "letmein"
jim = login_as :email => "jim@jim.com",
               :password => "letmein"
jim.add_to_shopping_cart 1.copy, "Groovy In Action"
jim.checkout
```

Submission 642 © ASERT 2007



… DSLs …

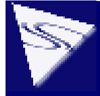
- **How to write your DSL**
  - **Pair with your customer**
  - **Explore the syntax of your DSL using actual inputs and outputs**
  - **Use the syntax checker to ensure your code is syntactically correct**
  - **Don’t over-engineer**
  - **Then implement it**

Submission 642 © ASERT 2007

… DSLs …

- **Ruby DSL Toolkit**
  - Open classes enable methods on literals
  - Method parenthesis are optional
  - Named parameters, using hashes and symbols
  - Code blocks to nest statements about something
  - Advanced features: Metaprogramming

Submission 642 © ASERT 2007



… DSLs …

- **Ruby Metaprogramming**
  - **eval, method_missing, define_method**
- **Groovy**
  - **invokeMethod, getProperty, setProperty**
- **These techniques may seem hackish at first, but they actually eliminate duplication, and become natural**

Submission 642 © ASERT 2007

… DSLs

- **Summary**
  - **Enable your customer to write tests as simple phrases, using the terminology of the business**

Submission 642 © ASERT 2007

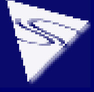
Refactoring changes

```
class Person {
    private name, age
    Person(name, age) {
        this.name = name
        this.age = age
    }
    def haveBirthday() { age++ }
    def describe() { "$name is $age years old" }
}

class StaffMemberUsingInheritance extends Person {
    private salary
    StaffMemberUsingInheritance(name, age, salary) {
        super(name, age)
        this.salary = salary
    }
    def describe() {
        super.describe() + " and has a salary of $salary"
    }
}

class StaffMemberUsingDelegation {
    private delegate
    private salary
    StaffMemberUsingDelegation(name, age, salary) {
        delegate = new Person(name, age)
        this.salary = salary
    }
    def haveBirthday() {
        delegate.haveBirthday()
    }
    def describe() {
        delegate.describe() + " and has a salary of $salary"
    }
}
```

```
…
class StaffMemberUsingMOP {
    private delegate
    private salary
    StaffMemberUsingMOP(name, age, salary) {
        delegate = new Person(name, age)
        this.salary = salary
    }
    def invokeMethod(String name, args) {
        delegate.invokeMethod(name, args)
    }
    def describe() {
        delegate.describe() + " and has a salary of $salary"
    }
}

def p1 = new StaffMemberUsingInheritance('Tom', 20, 1000)
def p2 = new StaffMemberUsingDelegation('Dick', 25, 1100)
def p3 = new StaffMemberUsingMOP('Harry', 30, 1200)
p1.haveBirthday()
println p1.describe()
p2.haveBirthday()
println p2.describe()
p3.haveBirthday()
println p3.describe()
```

```
Tom is 21 years old and has a salary of 1000
Dick is 26 years old and has a salary of 1100
Harry is 31 years old and has a salary of 1200
```

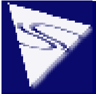
Submission 642 © ASERT 2007



Aspects vs Meta Programming

```
class TimingInterceptor extends TracingInterceptor {
    private beforeTime
    def beforeInvoke(object, String methodName, Object[] arguments) {
        super.beforeInvoke(object, methodName, arguments)
        beforeTime = System.currentTimeMillis()
    }
    public Object afterInvoke(Object object, String methodName, Object[] arguments, Object result) {
        super.afterInvoke(object, methodName, arguments, result)
        def duration = System.currentTimeMillis() - beforeTime
        writer.write("Duration: $duration ms\n")
        writer.flush()
        return result
    }
}

def proxy = ProxyMetaClass.getInstance(util.CalcImpl.class)
proxy.interceptor = new TimingInterceptor()
proxy.use {
    assert 7 == new util.CalcImpl().add(1, 6)
}
// =>
// before util.CalcImpl.ctor()
// after  util.CalcImpl.ctor()
// Duration: 0 ms
// before util.CalcImpl.add(java.lang.Integer, java.lang.Integer)
// after  util.CalcImpl.add(java.lang.Integer, java.lang.Integer)
// Duration: 16 ms
```

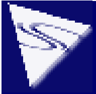
Submission 642 © ASERT 2007



Dependency Injection vs Meta Progamming …

- **Dependency Injection?**
  - **Central configuration of service objects**
  - **Transparent injection of dependent service objects into service objects**
  - **Service objects are instantiated by the dependency injection framework**
  - **Improves testability**
    - ***no explicit binding to dependencies***

Submission 642 © ASERT 2007



… Dependency Injection vs Meta Progamming …

- **Improves testability**
  - **no explicit binding to dependencies?**

```
class PrintService {
    private printer =
            new PhysicalPrinter('Canon i9900')
}
class PrintServiceTest extends GroovyTestCase {
    def testPrintService() {
        def printService = new PrintService()
        printService.print()
        // go to the printer and fetch the page
    }
}
```

```
class PrintService
    def initialise
        @printer =
            PhysicalPrinter.new('Canon i9900')
    end
end
class PrintServiceTest < Test::Unit
    def test_print_service
        print_service = PrinterService.new
        print_service.print
        # go to the printer and fetch the page
    end
end
```

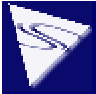
Submission 642 © ASERT 2007

- **Improves testability**

```groovy
class TestablePrintService {
    def printer
}
class TestablePrintServiceTest extends GroovyTestCase {
    def testPrintService() {
        def printService = new TestablePrintService()
        printService.printer = new StubPrinter()
        printService.print()
        //...
    }
}
```

```ruby
class TestablePrintService
    attr_accessor :printer
end
class TestablePrintServiceTest < Test::Unit::TestCase
    def test_print_service
        print_service = TestablePrintService.new
        print_service.printer = StubPrinter.new
        print_service.print
        #...
    end
end
```

Submission 642 © ASERT 2007



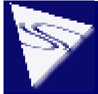
… Dependency Injection vs Meta Progamming …

- **Improves reuse?**

```
class UnreusablePrintService
    def initialise(printer_name)
        @printer = PostScriptPrinter.new(ip_address)
    end
    # printer logic below
    # ...
end
# What if I don't want to use a Physical Printer?
```

```
class ReusablePrintService
    attr_accessor :printer
    # interesting printer logic below
    # ...
end
# One customer may use a network PostScript printer
class PostScriptPrinter
    # ...
end
# The next customer may use a local Windows printer.
class LocalWindowsPrinter
    # ...
end
# No change is required to the ReusablePrintService!
```

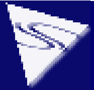
Submission 642 © ASERT 2007



… Dependency Injection vs Meta Progamming …

- **Downsides.....**
  - **Code becomes more complex**
  - **There is some magic**
- **Do I really need this?**
  - **In Java, the consensus is yes!**
  - **But Ruby is extremely dynamic**
  - **We can change classes at runtime**

Submission 642 © ASERT 2007

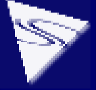


… Dependency Injection vs Meta Progamming

- **Printer dependency can be redefined in test code**
- **Benefits of Dependency Injection only become apparent in larger projects**
  - **where we need lots of reuse and centralized configuration of services**
- **In smaller projects,this is just a burden**

```
# Create stub/mock services in a separate module
Printer = Stubs::Printer
class PrintServiceTest < Test::Unit
  def test_print_service
    # The Printer instantiated by this class
    # is now a stub!
    print_service = UntestablePrintService.new
    print_service.do_something_interesting_with_printer
  end
end
```

Submission 642 © ASERT 2007

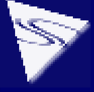


Checked vs unchecked exceptions

- **From [1]: "*Statically checked exceptions lead to leaking abstractions. This happens when a component interacts with another component that declares to throw unrelated exceptions that stem from a third component but which can only be handled on a higher level*."**

Submission 642 © ASERT 2007

- Groovy auto converts checked exceptions into unchecked
- Techniques such as autoboundaries can be used to overcome issues with existing code that abuses checked exceptions



Further Information

- [1] *Dynamic vs. Static Typing — A Pattern-Based Analysis*, Pascal Costanza, University of Bonn, 2004
  http://p-cos.net/documents/dynatype.pdf
- [2] *Interface-Oriented Design*, Ken Pugh, Pragmatic Programmers, 2006
- [3] Bruce Eckel. *Does Java need Checked Exceptions?*.
  www.mindview.net/Etc/Discussions/CheckedExceptions
- [4] Kevlin Henney. *Null Object*. EuroPLoP 2002, Proceedings.

Submission 642 © ASERT 2007