The Craftsman: 51
Brown Bag VIII
Ruby Visitor

Robert C. Martin
2 Aug, 2007

December 1, 1944

The November elections resulted in a landslide victory for Henry Wallace against Dewey. The surging economy, and the near bloodless thwarting of the Axis invasion of Mexico were huge factors in his popularity. Dewey had tried to use the frightening growth of the national debt as a platform for fiscal restraint, but the people weren't in a mood to hear about belt-tightening.

The general public was not yet aware of the Turing's conclusion about Clyde. They did not know that they had a little over 14 years to live. But there had been enough leaks, rumors, and innuendos that it was clear that something big *was up. The atmosphere in the nation felt like the hour before a big storm.*

Wednesday, 6 Mar 2002, 11:00

I'm not sure why they call these meetings "Brown bags". Apparently it's something from the old days at Los Alamos. In any case, over 30 people filed into the conference room for this one. The speaker was someone I hadn't met before. His name was Dave Chelimsky.

As I walked into the room I saw Avery and Jean sitting together in the back. Avery's expression was flat, and he made no eye-contact. Jasmine and Jerry were on the other side of the room, also in the back, talking intensely and happily about something. Jasper was close to the front, clearly anticipating this talk. Not surprisingly there was an empty seat next to him. I considered leaving it empty, but I had a notion that sitting next to him could be interesting today.

The rear wall was populated by about a half dozen virtual participants. Their faces stared out from the windows that held their virtual session. This was a first. We hadn't had virtual participants at any of our previous brown bags.

The room grew silent and the speaker began. Chelimsky's style might be described as intense nonchalance. It was clear that he cared deeply about his topic; but his demeanor was almost apologetic for his passion – as if to say: "I care about this, but I understand if you don't."

"I understand that you have been talking about Ruby lately." He stared out at us a smiled as if Ruby were a swear word that everyone was shocked to hear. "I thought it would be interesting if we looked at a common design pattern in both Java and Ruby, and compared the differences. And what better design pattern to study than Visitor?" Again, that smile, as though he was sharing some inside joke with us.

"The Visitor pattern can be really useful when you want the decoupling of polymorphism without the coupling of inheritance. Consider, for example, a simply hierarchy of employees. Perhaps we have an `Employee` base class that holds the name of the employee. Perhaps we also have two derivatives named `HourlyEmployee` and `SalariedEmployee`. `HourlyEmployee` holds a list of `TimeCard` objects, whereas `SalariedEmployee` holds just the salary. Now let's say we want to generate a report, one line per employee. For every hourly employee we want that line to read 'Hourly Bob worked 8 hours.', wheras

for every salaried employee we'd like it to read 'Salared Bill earns $350.'"

"Clearly we could put an abstract function named `reportLine` into the `Employee` base class and implement it appropriately in the two derivatives. However…" And with this he stared out at us meaningfully, "as you all know this would violate the Single Responsibility Principle!" He continued to stare meaningfully.

I understood him. I looked around and could tell that Jasper, Jerry, Jasmine, Jean, Adelade, and even Avery did too. But there were quite a few people in that room who looked uncomfortable. Was it possible that they did not know about the SRP? Was the SRP something related to the tight group under the sway of Jean?

David reared back a bit and nodded in a way that made it clear that he had not really expected everyone to know about the SRP. "It violates the SRP because the Employee hierarchy should not know about the format of a report. Classes that deal with business rules should not also be coupled to things like report formats!"

"So, then, what do we do? How do we get the polymorphic behavior we want without coupling it to the Employee hierarchy? The answer is the Visitor pattern!"

And with that he began to type on the front wall. Or rather he wiggled his fingers as *though* he had told the wall to track his typing. It was clear, however, as the characters rapidly flew in from the left and one-by-one took their place on the wall, that his was all prepared, and just a bit of theatre. The whole typing charade took less than 5 seconds.

```
public class ReportVisitorTest {
  private HourlyEmployee hal;
  private SalariedEmployee sam;
  private EmployeeReportVisitor v;

  @Before
  public void createEmployees() {
    hal = new HourlyEmployee("Hal");
    hal.addTimeCard(new TimeCard(8));

    sam = new SalariedEmployee("Sam", 500);
    v = new EmployeeReportVisitor();
  }

  @Test
  public void hourlyEmployeeReportsHours() {
    Employee e = hal;
    e.accept(v);
    assertEquals("Hourly Hal worked 8 hours.", v.getReportLine());
  }

  @Test
  public void salariedEmployeeReportsSalary() {
    Employee e = sam;
    e.accept(v);
    assertEquals("Salaried Sam earns $500.", v.getReportLine());
  }
}
```

"Consider this unit test." He said. "It clearly exhibits the polymorphism we want. And it uses a visitor to make sure that the polymorphic behavior is decoupled from the `Employee` hierarchy. Here are the rest of the classes."

With that he waved his hand at the screen, and the rest of the code flashed into place on the screen with a resounding clash of cymbals the opening guitar lead from *Layla*.

I looked around and caught a few eyes that were saying to me, and to each other "What a ham."

```
public abstract class Employee {
```

```
  protected String name;

  public Employee(String name) {
    this.name = name;
  }

  public abstract void accept(EmployeeVisitor v);

  public String getName() {
    return name;
  }
}
```

```
public class SalariedEmployee extends Employee {
  private int salary;

  public SalariedEmployee(String name, int salary) {
    super(name);
    this.salary = salary;
  }

  public void accept(EmployeeVisitor v) {
    v.visit(this);
  }

  public int getSalary() {
    return salary;
  }
}
```

```
public class HourlyEmployee extends Employee {
  private List<TimeCard> timeCards;
  public HourlyEmployee(String name) {
    super(name);
    timeCards = new ArrayList<TimeCard>();
  }

  public void addTimeCard(TimeCard timeCard) {
    timeCards.add(timeCard);
  }

  public void accept(EmployeeVisitor v) {
    v.visit(this);
  }

  public int getHours() {
    int hours = 0;
    for (TimeCard tc : timeCards)
      hours += tc.getHours();
    return hours;
  }
}
```

```
public class TimeCard {
  private int hours;

  public TimeCard(int hours) {
    this.hours = hours;
  }

  public int getHours() {
    return hours;
  }
}
```

```
public interface EmployeeVisitor {
```

```
  void visit(HourlyEmployee hourlyEmployee);
  void visit(SalariedEmployee salariedEmployee);
}
```

```
public class EmployeeReportVisitor implements EmployeeVisitor {
  private String reportLine;

  public String getReportLine() {
    return reportLine;
  }

  public void visit(HourlyEmployee hourlyEmployee) {
    int hours = hourlyEmployee.getHours();
    String name = hourlyEmployee.getName();
    reportLine = String.format("Hourly %s worked %d hours.", name, hours);
  }

  public void visit(SalariedEmployee salariedEmployee) {
    String name = salariedEmployee.getName();
    int salary = salariedEmployee.getSalary();
    reportLine = String.format("Salaried %s earns $%d.", name, salary);
  }
}
```

I read through the code for a few seconds and saw the unmistakable structure of the Visitor pattern.

David spun around to face the audience and said: "Notice how the report formatting code is sequestered nicely in the `EmployeeReportVisitor` class. Notice also that the Employee hierarchy knows nothing about it. This is nice, and conforms well with the SRP. " But then he took on that apologetic demeanor of his, he let his shoulder's drop and his arms go limp. He said: "But, there's a problem. See how the `EmployeeVisitor` names the two derivatives of `Employee` as arguments of its two `visit` functions? This creates a cycle of dependencies that makes it hard to add new `Employee` derivatives. "

I'd been through this before, so it wasn't a surprise to me; but others seemed puzzled.

"Look!" he said with faux frustration, "The `Employee` class depends on `EmployeeVisitor` which depends upon both `SalariedEmployee` and `HourlyEmployee`. If you add a new derivative of `Employee`, like `CommissionedEmployee`, you have to add a new `visit` function to `EmployeeVisitor`. Since `Employee` depends on `EmployeeVisitor`, this change will affect `Employee` and anyone who uses `Employee`!"

Other than Jerry, Jasmine, and my other immediate co-workers, I didn't see any lightbulbs go on. Did these people *really* not understand transitive dependencies?

David stood there for a minute, obviously hoping to get some feedback from the audience that they understood the issue. I saw Jean flash him a meaningful look, and then he dropped his hands and went on.

"There have been a number of solutions to this problem with Visitor." He said. We can sometimes use a Decorator, or an Extension Object, or even an AcyclicVisitor. But each of those have their downsides. But now look at the same problem in Ruby."

This time David brandished his arms at the screen as if invoking a magic spell.

```
describe EmployeeReportVisitor do
  before do
    @hal = HourlyEmployee.new("Hal")
    @hal.addTimeCard(TimeCard.new(8))
    @sam = SalariedEmployee.new("Sam", 500)
    @v = EmployeeReportVisitor.new
  end

  it "should generate hourly report for hourly employee" do
    @hal.accept(@v)
    @v.getReportLine.should == "Hourly Hal worked 8 hours."
```

```
  end

  it "should generate salaried report for salaried employee" do
    @sam.accept(@v)
    @v.getReportLine.should == "Salaried Sam earns $500."
  end
end
```

"Once again, here are the unit tests – written in *rspec*. You can clearly see the similarity. You can see that we are expecting polymorphic behavior through the agency of a visitor. Now here is the rest of the code!" The wall filled with Ruby code while the last few strains of the *Also Sprach Zarathustra* overture blared out.

```
class Employee
  def initialize(name)
    @name = name;
  end

  def getName
    @name
  end
end
```

```
class HourlyEmployee < Employee
  def initialize(name)
    super(name)
    @timeCards = []
  end

  def addTimeCard(timeCard)
    @timeCards << timeCard;
  end

  def accept(visitor)
    visitor.visitHourly(self)
  end

  def getHours
    hours = 0;
    @timeCards.each {|tc| hours += tc.getHours}
    hours
  end
end
```

```
class SalariedEmployee < Employee
  def initialize(name, salary)
    super(name)
    @salary = salary
  end

  def getSalary
    @salary
  end

  def accept(visitor)
    visitor.visitSalaried(self)
  end
end
```

```
class TimeCard
  def initialize(hours)
```

```
    @hours = hours
  end

  def getHours
    @hours
  end
end
```

```
class EmployeeReportVisitor
  def visitHourly(hourlyEmployee)
    name = hourlyEmployee.getName
    hours = hourlyEmployee.getHours
    @reportLine = "Hourly #{name} worked #{hours} hours."
  end

  def visitSalaried(salariedEmployee)
    name = salariedEmployee.getName
    salary = salariedEmployee.getSalary
    @reportLine = "Salaried #{name} earns $#{salary}."
  end

  def getReportLine
    @reportLine
  end
end
```

"Again, you should be able to see the similarity. But notice! There is no `EmployeeVisitor` base class! The `accept` methods simply call `visitHourly` or `visitSalaried` on some object of unknown type. In our case it just happens to be an `EmployeeReportVisitor` object. The ruby runtime works this out. So there is no `EmployeeVisitor` to change if we add a new derivative of `Employee`. And even if there were, notice that `Employee` does not depend upon `EmployeeVisitor`. Indeed, there is no `accept` method declared in the `Employee` base!"

Jerry raised his hand. David stopped and called on him.

"This makes my head hurt." Said Jerry. How can the Ruby compiler generate any code? How can it allow `HourlyEmployee.accept` to call `visitHourly` on some `visitor` object of uknown type?"

"First, Jerry, there is no Ruby compiler! The language is interpreted. So there is no code to generate. But even if there were, the compiler wouldn't have any trouble. It would simply ask the `visitor` object to execute the `visitHourly` method. If that `visitor` object did not have that method, it would throw an exception. "

"That makes my head hurt even more." Said Jerry as he sat down again.

David continued. "Can you see that the dependency cycle is broken? This is one of the great benefits of dynamically typed languages. The dependency knots created in programs written in statically typed languages simply don't exist. They are resolved at runtime instead of compile time!"

Predictably, Jasmine stood up and said: "Yes, but isn't this inherently unsafe? How can you defer your errors to runtime without risk that your system will crash in production?"

"How could it be any less safe if we use Test Driven Development? If we write our unit tests according to Mr. C's rules, then we'll find the type errors that the Java compiler would have found." He stopped for a second and then continued. "Let me put it this way. You have to admit that if we are using TDD, then the risk of a type error getting through is very small. Yet the benefits of dynamic typing are huge. Code is smaller, simpler, and more flexible. You have fewer dependency knots. It is easier to make changes. So, if you are using TDD, the benefits vastly outweigh the risks."

Jasmine looked skeptical. Jerry was befuddled. And Jean was glancing at her watch.

"I think it's time to go, Dears." She said. And so we all gathered up our things and filed out.

As I walked out of the room I realized that Jasper had remained utterly silent the whole time. I looked back and saw him behind me. He beamed at me with that huge toothy smile of his.