Extracted from:

The ThoughtWorks Anthology 2

More Essays on Software Technology and Innovation

This PDF file contains pages extracted from *The ThoughtWorks Anthology 2*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printer versions; the content is otherwise identical.

Copyright © 2012 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The ThoughtWorks® Anthology 2

More Essays on Software Technology and Innovation

Edited by Michael Swaine

The ThoughtWorks Anthology 2

More Essays on Software Technology and Innovation

Farooq Ali
Ola Bini
Brian Blignaut
James Bull
Neal Ford
Martin Fowler
Luca Grulla
Alistair Jones
Aman King
Patrick Kua
Marc McNeill
Julio Maia
Mark Needham
Sam Newman
Rebecca Parsons
Cosmin Stejerean

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

3.1 Objects over Classes?

What is a "objects over classes" mind-set?

Well, it's the presiding idea that comes out of thoughts around "objects above classes" in Java, "classes as objects" in Ruby, and "objects in lieu of classes" in JavaScript.

Consider a software solution to a business problem. Irrespective of how you approach it, the solution will ultimately take the form of an application that needs a runtime environment to function. Multiple instructions get executed within this environment, and combinations of these instructions lead to the business problem getting solved.

As a programmer, you need to understand this environment. After all, you're the one who starts off everything, stating what may happen, when it may happen, and how it may happen.

This is where the *object-oriented paradigm* comes in. It gives you a mental model to envision how things take place within the runtime environment. You see the entire system as an ecosystem where various *objects* interact with each other. These objects are entities that tell other entities what to do, react to other entities, change their states, are either introverted or chatty, and are short-lived or sometimes immortal.

Not everyone looks at the runtime environment in the same way: alternate views include paradigms such as procedural, functional, event-driven, and so on. However, the OO paradigm is, by far, the most popular. It is implemented by doing *object-oriented programming* using *object-oriented languages*. These languages provide constructs for specifying object behavior and state.

OO languages can differ in the representational model they allow programmers to use, although there are more commonalities than differences. It's only when you come across a stark distinction that you feel like taking a step back to *think*.

Besides the object, an important concept in OO is the *class*. It's a construct provided by most OO languages to define a template or a blueprint for creating objects.

How do these traditional class constructs help?

At a high level, they allow you to see an unchanging view of the system in one shot, as a network of interconnected classes. They can be connected by inheritance, composition, or collaboration.

• Click HERE to purchase this book now. discuss

At a more focused level, classes represent individual domain concepts, capturing data attributes and operations. Often classes share common properties that are pulled into an inherited generalization. This creates a taxonomy that is observable in the real world.

At a low level, classes define the contract for their instances' interactions, usually accompanied with implementation.

But what's the role of a class in a running application? Once kick-started, almost all heavy lifting is done by objects within the environment. Classes play a passive role, acting only as reference templates for creating new instances. An exception to this is a class with class-level methods that get invoked during runtime.

Classes, as described earlier, play a significant role mostly during an application's conceptualization and construction. As such, I see them as a designing tool for capturing a static representation of the system. They are of limited use during runtime.

Trygve Reenskaug [RWL95], creator of the MVC pattern and contributor to the UML modeling language, once had this to say about the topic:

> *"The class/object duality is essential to both object oriented programming and to object oriented modeling."*

This distinction and the impact of recognizing this duality is what I hope to cover. If we agree that working software matters the most, can't we say that the runtime environment is as important as its static representation? As we design and implement, shouldn't we weigh object-related considerations, in addition to and sometimes over class-related considerations?

3.2 Class Focus vs. Object Focus

Let's begin by differentiating between a class-focused approach and an object-focused approach.

A *class-focused approach* is what I deem to be driven by the goal of modeling the domain in a way the business will recognize it. It does not explicitly track how domain concepts interplay over the course of time but attempts to cover every detail in a single static snapshot. A typical way to come up with a class-based solution is to mark all nouns in a given problem statement and to make classes for each of them. Generalizations and specializations figure their way in as an inheritance hierarchy. A typical example from an enterprise domain will have classes like `Person`, `Employee`, `Manager`, `SeniorManager`, and so on.

• Click HERE to purchase this book now. discuss

In an *object-focused approach*, the possible runtime relationships and interactions of an object become the driving factor behind the design. It leads to representations of the various roles a domain concept can play at different points in time. There may be an overlap of these roles between multiple domain entities. For example, roles in an enterprise domain include Billable, NonBillable, Delegator, Approver, Reportee, and so on.

The Role of Roles

Let's take our example further. In an enterprise, it'd be intuitive to assume that a Senior Manager is a specialization over a Manager, who is an Employee, who obviously is a Person. However, in the runtime environment, let's say on a typical day in office, do you see your Senior Manager *visibly* made up of individual parts, some beaming down from a phantom of an Employee and some from a ghostly Person? That'd be funny if all too apparent. In reality, the Senior Manager is just one complete being, who is performing many important roles. If you need approval of a proposal bid, she'd be the *approver*. If you need to report your timesheet, she'd be your *reporter*. On days when she's on leave, she'd *delegate* such tasks to a Manager who'd easily substitute into the roles of approver and reporter for you.

If the focus isn't on roles, the class structure for the previous domain may look like the following in Java:

```
class Person {
  // ...
}
class Employee extends Person {
  // ...
}
class Manager extends Employee {
  // ...
}
class SeniorManager extends Manager {
  // ...
}
class SalesAssociate extends Employee {
  // ...
}
```

If, however, we were to represent roles that the domain entities could play, we might end up with this:

```
interface Approver {
  // ...
}
interface Delegator {
```

```
  // ...
}
interface ProposalWriter {
  // ...
}
class SeniorManager implements Approver, Delegator {
  // ...
}
class Manager implements Approver {
  // ...
}
class SalesAssociate implements ProposalWriter {
  // ...
}
```

Notice how every role is represented by a *role interface*.[1] All the domain entity classes like SeniorManager, Manager, and so on, implement these roles as appropriate. Also note the complete lack of inheritance. So, how is code reused? Surely there must be similarities between how a Senior Manager or a Manager performs approvals.

In the runtime environment, where inheritance doesn't play a *visible* role, inheritance is reduced to a technique for reusing code and avoiding duplication. But there are other ways of achieving reuse, one being composition with delegation. So, perhaps composition may be an alternative to inheritance.[2] Let's see how.

```
interface Approver {
  ApprovalDecision decideApproval(ApprovalRequest approvalRequest);
}
class MediumLevelApprovalStrategy implements Approver {
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    // ... some business decision rules
    return approvalDecision;
  }
}
class LowLevelApprovalStrategy implements Approver {
  // ...
}
class SeniorManager implements Approver, Delegator {
  Approver approvalStrategy = new MediumLevelApprovalStrategy();
  public SeniorManager(String name) {
    // ...
  }
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    return approvalStrategy.decideApproval(approvalRequest);
```

1. http://martinfowler.com/bliki/RoleInterface.html
2. http://c2.com/cgi/wiki?DelegationIsInheritance

```
  }
  // ...
}
class Manager implements Approver {
  Approver approvalStrategy = new LowLevelApprovalStrategy();
  public Manager(String name) {
    // ...
  }
  public ApprovalDecision decideApproval(ApprovalRequest approvalRequest) {
    return approvalStrategy.decideApproval(approvalRequest);
  }
  // ...
}
```

As seen, the domain entities instantiate a strategy of their choice and then trust it to do the decision making for them. In this case, the Manager uses a low-level approval strategy, while the Senior Manager uses a medium-level strategy. This approach makes everything very explicit and gets away from complexities of selective method overrides, template methods, private vs. protected access, and so on, which are all part and parcel of complex inheritance structures.

An added benefit is that the reuse happens at runtime. It is trivial to change existing dependencies as the live situation changes. For example, the Senior Manager based on certain conditions could change her approval strategy to be low level for some time, only to later go back to a medium-level strategy. This can't be easily achieved with inheritance where the implementation is locked down at compile time.

Now let's consider another example: the problem statement is of generating a voter list. Voters can be people who are 18 years or older and who are nationals of a certain country. To introduce complexity, we'll state that voters can also be countries that can vote within councils of nations that they're members of. Given the domain entities, we can have classes like these:

• Click HERE to purchase this book now. discuss

ObjectsOverClasses/java/voterlist/classbased/VoterListClassBased.java

```
class CouncilOfNations {
    private Collection<Country> memberNations;
    public CouncilOfNations(Collection<Country> memberNations) {
        this.memberNations = memberNations;
    }
    public boolean contains(Country country) {
        return memberNations.contains(country);
    }
}

class Country {
    private String name;
    public Country(String name) {
        this.name = name;
    }
}

class Person {
    private String name;
    private int age;
    private Country country;
    public Person(String name, int age, Country country) {
        // ...
    }
    public boolean canVoteIn(Country votingJurisdiction) {
        return age >= 18 && votingJurisdiction.equals(country);
    }
}

abstract class AbstractVoterList<T, X> {
    private Collection<T> candidateVoters;
    public AbstractVoterList(Collection<T> candidateVoters) {
        this.candidateVoters = candidateVoters;
    }
    public Collection<T> votersFor(X votingJurisdiction) {
        Collection<T> eligibleVoters = new HashSet<T>();
        for (T voter : candidateVoters) {
            if (canVoteIn(voter, votingJurisdiction)) {
                eligibleVoters.add(voter);
            }
        }
        return eligibleVoters;
    }
    protected abstract boolean canVoteIn(T voter, X votingJurisdiction);
}
class PersonVoterList extends AbstractVoterList<Person, Country> {
    public PersonVoterList(Collection<Person> persons) {
        super(persons);
    }
```

```
    protected boolean canVoteIn(Person person, Country country) {
        return person.canVoteIn(country);
    }
}
class CountryVoterList extends AbstractVoterList<Country, CouncilOfNations> {
    public CountryVoterList(Collection<Country> countries) {
        super(countries);
    }
    protected boolean canVoteIn(Country country,
                                CouncilOfNations councilOfNations) {
        return councilOfNations.contains(country);
    }
}
```

The previous can be invoked like this:

ObjectsOverClasses/java/voterlist/classbased/VoterListClassBased.java

```
Country INDIA = new Country("India");
Country USA = new Country("USA");
Country UK = new Country("UK");
Collection<Person> persons = asList(
    new Person("Donald", 28, INDIA),
    new Person("Daisy", 25, USA),
    new Person("Minnie", 17, UK)
);
PersonVoterList personVoterList = new PersonVoterList(persons);
System.out.println(personVoterList.votersFor(INDIA)); // [ Donald ]
System.out.println(personVoterList.votersFor(USA));   // [ Daisy ]
Collection<Country> countries = asList(INDIA, USA, UK);
CountryVoterList countryVoterList = new CountryVoterList(countries);
CouncilOfNations councilOfNations = new CouncilOfNations(asList(
    USA, INDIA
));
System.out.println(countryVoterList.votersFor(councilOfNations));
// [ USA, India ]
```

If we were to convert the previous into a more object-focused solution, we need to start by looking at important interaction points and come up with new names for the collaborators according to what they're doing, rather than what they are. Once identified, we can use refactorings like *Extract Interface*, *Pull Up Method*, and *Rename Method* to change the code structure to look something like this:

ObjectsOverClasses/java/voterlist/rolebased/VoterListRoleBased.java

```
interface VotingJurisdiction {
    boolean covers(VotingJurisdiction votingJurisdiction);
}
interface Voter {
    boolean canVoteIn(VotingJurisdiction votingJurisdiction);
}
```

```
class CouncilOfNations implements VotingJurisdiction {
    private Collection<Country> memberNations;
    public CouncilOfNations(Collection<Country> memberNations) {
        this.memberNations = memberNations;
    }
    public boolean covers(VotingJurisdiction votingJurisdiction) {
        return this.equals(votingJurisdiction) ||
            memberNations.contains(votingJurisdiction);
    }
}
class Country implements VotingJurisdiction, Voter {
    private String name;
    public Country(String name) {
        this.name = name;
    }
    public boolean covers(VotingJurisdiction votingJurisdiction) {
        return this.equals(votingJurisdiction);
    }
    public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
        return votingJurisdiction.covers(this);
    }
}
class Person implements Voter {
    private String name;
    private int age;
    private Country country;
    public Person(String name, int age, Country country) {
        // ...
    }
    public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
        return age >= 18 && votingJurisdiction.covers(country);
    }
}
class VoterList {
    private Collection<Voter> candidateVoters;
    public VoterList(Collection<Voter> candidateVoters) {
        this.candidateVoters = candidateVoters;
    }
    public Collection<Voter> votersFor(VotingJurisdiction votingJurisdiction) {
        Collection<Voter> eligibleVoters = new HashSet<Voter>();
        for (Voter voter : candidateVoters) {
            if (voter.canVoteIn(votingJurisdiction)) {
                eligibleVoters.add(voter);
            }
        }
        return eligibleVoters;
    }
}
```

The usage remains very similar to the previous case, except that both personVoterList and countryVoterList will be instances of the same class: VoterList.

The former solution effectively uses inheritance combined with generics and the template method design pattern. The latter solution is implemented using role interfaces, which ends up needing neither generics nor any apparent design pattern.

There are further differences between the two solutions.

In the first approach, PersonVoterList allows only for checking a Person instance against a Country jurisdiction, and CountryVoterList checks only between a Country and a CouncilOfNations.

In the second approach, VoterList can check any kind of Voter implementation against any kind of VotingJurisdiction implementation. For example, personVoterList.votersFor(councilOfNations) returns a consolidated list of [Donald, Daisy] across the countries [INDIA, USA].

The philosophy behind the latter approach is that when it comes to interactions, what matters is not what *type* of object an argument is (as in Person, Country, or CouncilOfNations) but whether the object can play an expected role or not (like Voter and VotingJurisdiction).

The former approach creates a restrictive system where an object is highly constrained in its choices of what other objects it can interact with. With role interfaces, an object can be more *promiscuous* with respect to collaboration. Objects will be expecting only subsets of interfaces to be satisfied at a time, allowing many different types of objects to satisfy these minimal interfaces. Determining whether the interaction itself makes sense is left to the consuming code, which can be validated by unit tests.

With an object-focused mind-set, you should be thinking more in terms of role interfaces as opposed to header interfaces. A *header interface*[3] helps only in specifying a complete contract for a class. It discourages the notion of instances taking on any behavior not covered by the interface. It also places strict demands on substitute implementations that will need to satisfy the complete interface or nothing at all.

Header interfaces are prevalent in projects where programmers make a habit out of what Martin Fowler describes as “the practice of taking every class and pairing it with an interface.”[4]

3. http://martinfowler.com/bliki/HeaderInterface.html
4. http://martinfowler.com/bliki/InterfaceImplementationPair.html

• Click HERE to purchase this book now. discuss

Separation of Responsibilities

Another aspect to object focus is actively thinking about responsibilities that belong to a class vs. those of an object. Remember that in a live environment, objects are more active than classes. You should enhance these objects where possible instead of introducing class-level behavior. The question to ask is, "Should a class really be doing more than defining behavior of its instances?"

The following code is a *utility class*, something that quickly crops up in almost any project. These classes defy the expectation of acting as a blueprint and participate during runtime directly. They are usually stateless and hold multiple class-level *helper methods* that apply transformations on a set of inputs to return a desired output. These methods seem more procedural than OO.

As a project progresses, utility classes tend to bloat, sometimes to a scary extent! They can end up with all kinds of logic, ranging from seemingly harmless wrapper methods around primitive data types to critical domain-specific business rules. Bringing a rampant utility class back under control can be difficult. So, what can we do about functionality such as the following?

ObjectsOverClasses/java/utility/Utils.java

```
public class Utils {
    // ...
    public static String capitalize(String value) {
        if (value.length() == 0) return "";
        return value.substring(0, 1).toUpperCase() + value.substring(1);
    }
    // ...
    private static String join(List<? extends Object> values,
                               String delimiter) {
        String result = "";
        // ...
        return result;
    }
}
```

We can split these behaviors into individual classes with instance-level methods. Instances of core classes can then be decorated by these new wrapper classes.

ObjectsOverClasses/java/utility/Extensions.java

```
class ExtendedString {
    private String value;
    public ExtendedString(String value) {
        this.value = value;
    }
```

```
    public String toString() {
        return value;
    }
    public ExtendedString capitalize() {
        if (value.length() == 0) return new ExtendedString("");
        return new ExtendedString(
            value.substring(0, 1).toUpperCase() + value.substring(1)
        );
    }
}
class ExtendedList<T> extends ArrayList<T> {
    public ExtendedList(List<T> list) {
        super(list);
    }
    public String join(String delimiter) {
        String result = "";
        // ...
        return result;
    }
}
```

Here's the difference in the usages:

ObjectsOverClasses/java/utility/Utils.java

```
String name = Utils.capitalize("king"); // "King"

List<String> list = asList("hello", "world");
String joinedList = Utils.join(list, ", "); // "hello, world"
```

ObjectsOverClasses/java/utility/Extensions.java

```
ExtendedString extendedString = new ExtendedString("king");
String name = extendedString.capitalize().toString(); // "King"

ExtendedList<String> extendedList =
    new ExtendedList<String>(asList("hello", "world"));
String joinedList = extendedList.join(", "); // "hello, world"
```

Notice that in the former case, the functionality is globally accessible across the code base by virtue of being associated with a class. In the latter case, the functionality is available only if the consuming code has access to the right objects. This distinction is reminiscent of the global variables of yesteryear. There are similar risks[5] with class-level methods, especially if they modify class-level state.

I first came across the technique of decorating instances at runtime in Martin Fowler's *Refactoring [FBBO99]* book as a refactoring named *Introduce Local*

5. http://c2.com/cgi/wiki?GlobalVariablesAreBad

Extension. It's useful when you can't add behavior to a class directly because you don't have control over it, like Java's `String`.

With core data types such as integers, strings, and arrays, the problem may not remain restricted to class-level methods. If objects freely accept primitives as method parameters or are composed of the same, the objects may end up with misplaced responsibilities. A classic example in Java is using `Double` to represent money or using `String` for ZIP codes, telephone numbers, and email addresses. In most projects, such fields will be accompanied by special formatting or validation requirements. This is when we may struggle to find a place for corresponding logic. If the code doesn't end up in a utility class, it's likely to land in the object dealing with the primitive value. Neither is ideal. The *Refactoring [FBBO99]* book lists this *code smell* as *Primitive Obsession* and suggests refactorings like *Replace Data Value with Object* and *Replace Type Code with Class*.

Java's `Calendar` API is a real-world example of having numerous methods that accept and return primitive number values. Because of the resulting unwieldy API, Joda Time, a cleaner and more object-oriented alternative, has gained much popularity.[6]

Testing Perspective

There are two sides to testing with respect to an object focus.

On the one hand, test-driving your code, while having scenarios in mind instead of classes, can push you toward designs that are object-focused rather than class-focused. Since you're dealing with only one interaction at a time, you think about the roles of the objects in question rather than entire domain concepts. This may drive you to build your domain modeling gradually, evolving role after role. You may be surprised to see how many different domain entities suddenly fit into the roles you identify, which is something you may not catch otherwise.

On the other hand, if your solution has an object-focused design, its testability improves further. Techniques such as role interfaces make it especially easy to test collaborations . You may use a mocking framework, but trivially created stubs can suffice. Avoiding class-level methods further allows consuming code to invoke behavior via collaborators that are dependency-injectable, not hardwired. This makes mock substitutions possible.

6. http://joda-time.sourceforge.net

Here's how you can use stubs to unit test the role-based VoterList without needing to involve either Person or Country:

ObjectsOverClasses/java/voterlist/rolebased/VoterListTest.java

```
public class VoterListTest {
    @Test
    public void shouldSelectThoseWhoCanVote() {
        Voter eligibleVoter1 = new VoterWithEligibility(true);
        Voter eligibleVoter2 = new VoterWithEligibility(true);
        Voter ineligibleVoter = new VoterWithEligibility(false);
        Collection<Voter> candidateVoters = new HashSet<Voter>(asList(
            eligibleVoter1, ineligibleVoter, eligibleVoter2
        ));
        Collection<Voter> expectedVoters = new HashSet<Voter>(asList(
            eligibleVoter1, eligibleVoter2
        ));
        VoterList voterList = new VoterList(candidateVoters);
        assertEquals(expectedVoters,
            voterList.votersFor(new AnyVotingJurisdiction()));
    }

    static class VoterWithEligibility implements Voter {
        private boolean eligibility;
        public VoterWithEligibility(boolean eligibility) {
            this.eligibility = eligibility;
        }
        public boolean canVoteIn(VotingJurisdiction votingJurisdiction) {
            return eligibility;
        }
    }
    static class AnyVotingJurisdiction implements VotingJurisdiction {
        public boolean covers(VotingJurisdiction votingJurisdiction) {
            return true;
        }
    }
}
```

Steve Freeman, Nat Pryce, et al., share further thoughts on test-driving roles in their OOPSLA paper, *Mock Roles, Not Objects [FPMW04]*.

Code Base Indicators

Now that you've seen examples of class-focused solutions along with object-focused counterparts, will you be able to spot them in your own project? I'll share some signs that could help.

Do you use a framework that requires inheritance to work? A subclass is restricted in what it can reuse via inheritance. There are also constraints applied to its constructors. Such compile-time restrictions on the evolution

of a class indicate a lack of object-based thinking. Try to use framework-coupled classes only for plugging into the framework. Keep domain logic in separate classes that can change independently.

Do you have classes three or four levels deep in their inheritance hierarchy? Having multiple levels of inheritance means more and more behavior is accumulated at every level, leading to heavy objects that can do too many things. Question how much of the inherited behavior is actually relevant as per usage. Heavy objects are also difficult to unit test in isolation because they're hardwired to parent implementations. Even a *test-specific subclass*[7] will be cumbersome if you need to override multiple methods for stubbing.

Do you have more classes than you care for, especially those with only minor variations? Class explosion can be a sign of class-based thinking. Deep inheritance hierarchies result in too many classes. This worsens if parallel hierarchies exist. As an example, for every subclass of `Car`, say `Sedan` and `Hatchback`, we may need a corresponding `Chassis`, like `SedanChassis` and `HatchbackChassis`. At this point, we need to identify common roles across the classes and try composition instead of inheritance.

Do you have too few classes, each of them rather bloated? A system is healthy if there are multiple objects talking to each other, not if only a few introverted objects perform all the tasks. Having few classes indicates that responsibilities are distributed only among some objects during runtime. On top of that, if class-level methods are prevalent, classes are owning tasks that would otherwise be performed by objects. Note that having small independent classes as blueprints of objects with single responsibilities is different from having many deceptively small classes that inherit shared behavior and vary only marginally. The former is a positive sign of loose coupling, while the latter reflects high coupling.

There can be more of such indicators, but this list is a good starting point. Regular code introspection can be useful for a project: I've been part of teams that made it a practice, and it helped improve the quality of our code base.

7. http://xunitpatterns.com/Test-Specific%20Subclass.html