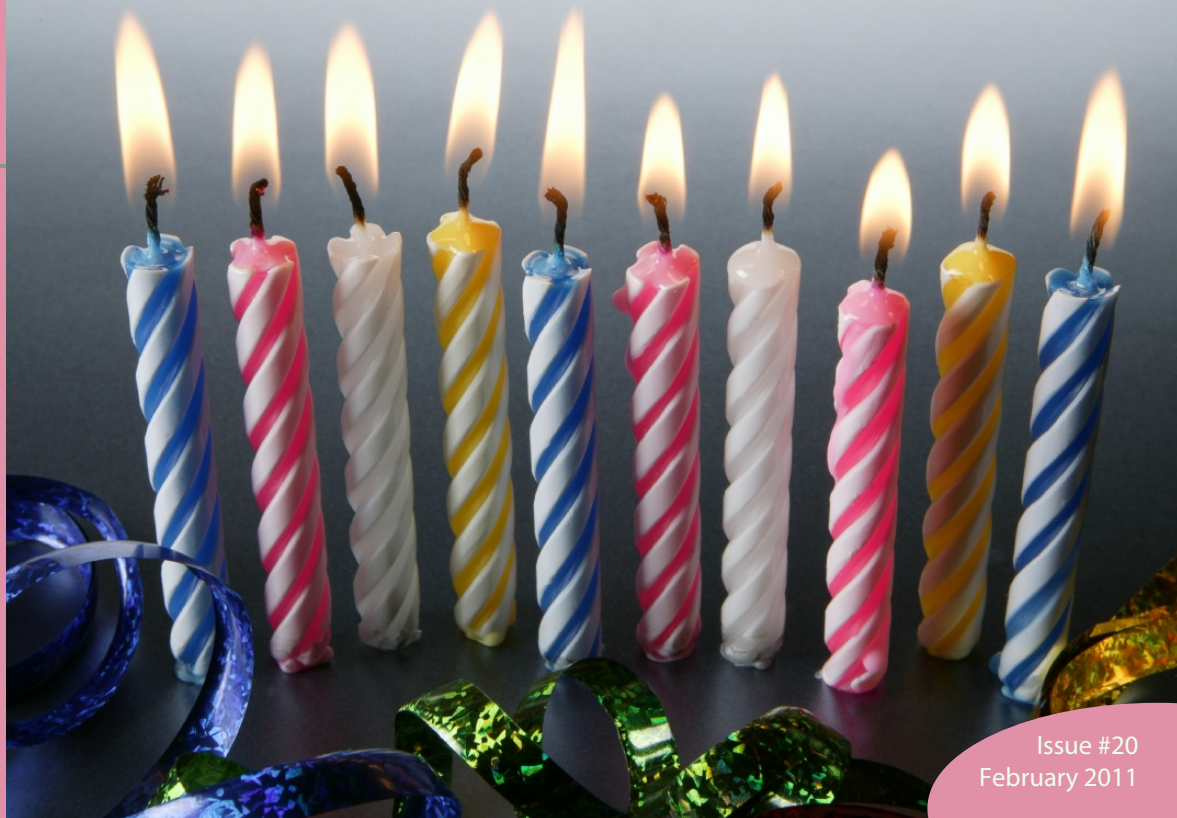
The Pragmatic Bookshelf

PragPub

The First Iteration



IN THIS ISSUE

Happy Birthday, Agile Manifesto!

Has it really been ten years? Time flies in two-week iterations.

Issue #20
February 2011

Contents

FEATURES

Up Front

The Agile Issue

by Michael Swaine

We're pleased to welcome you to this, our Agile issue, celebrating the tenth birthday of the Agile Manifesto, a document that changed the world of software development. To mark the occasion, we invited the authors of the Manifesto to share their thoughts. Ten of them agreed to do so. We think you'll find their reflections thought-provoking.

In this spirit of Agile reflection, we've also included a list of every Agile-themed article ever published in *PragPub*. They're all available and clickable, thus (you might say) adding 48 more Agile articles to this already-packed Agile issue.

In addition to these Agile reflections, this issue contains another article in the series on big ideas in software from Jeff Langr and Tim Ottinger, an article on "Refactoring Your Job" by Craig Riecke and a cool math article by Staffan Nöteberg. Then there's Jonathan Rasmusson's latest "Way of the Agile Warrior" column, our regular Choice Bits and Calendar departments, and the latest of John Shade's "Shady Illuminations," in which he writes a letter to the editor. The editor of *PragPub*, that is.

For the record, the 17 authors of the Agile Manifesto are: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas.

Abstraction

How to Tell a Cat from a Dog

by Tim Ottinger, Jeff Langr

There is a very deep antipathy between duplication and abstraction.

This continues our series on the four Big Ideas in software development. Be sure to check earlier issues for articles on Cohesion [U1] and Coupling [U2]. This month, our goal is to cast new light on *abstraction*.

A naive approach to object-oriented design is to create a system using classes that model real-world things. If you learned about object-oriented programming in the 1990s, chances are someone had you model Dog, Cat, Mammal, and Animal classes as an exercise. Abstraction meant implementing the parts that related to your software needs. You might have designed a bark() behavior, but not a tailWag() behavior, along with a few supporting attributes such as size and speed. Your classes were a straightforward abstraction of the real world, each with as many attributes and behaviors as made sense for that real-world element.

From this introduction to abstraction comes quite naturally a mindset that the best way to create an an object-oriented design is to model the real world, leaving a few bits out. This is not necessarily wrong, but it is misleading. Abstraction is deeper and more profound than this mindset makes it sound.

We base our primary focus on abstraction on a definition by Uncle Bob Martin: Abstraction is the *elimination of the irrelevant and the amplification of the essential*. See how we just emphasized both essential phrases in that definition and eliminated the BS about "the real world?"

Abstract and Concrete

We start our discussion of abstraction with the concept of abstract types vs. concrete types. Abstract types do not completely specify behavior, whereas concrete types contain specific code details for all behaviors. Purely abstract types in C#, Java, and the like (where absolutely no behavior is defined) are known as *interfaces*.

From concept to code, then, abstraction is directly implemented in the form of interfaces. The set of behaviors supported by a class appears as a standalone declaration, a contract of sorts:

```
public interface FineCalculator {
   BigDecimal charge(int daysLate);
}
```

The FineCalculator interface captures the concept of determining how much to charge library patrons for borrowed materials that they return late. The interface captures only this singular concept and no implementation details (other than the argument and return types). A fine calculator implementation will complete the interface by implementing charge(). You might imagine

BookFineCalculator, MovieFineCalculator, and NewReleaseFineCalculator implementations.

Though implementations may vary, the abstract concept of determining an appropriate fine charge for a given number of late days is likely to remain unchanged from the point of view of a FineCalculator user. Among the benefits of having this abstraction are:

- The specific portion of the client code that must obtain fines can be written once, regardless of the material types involved. "If" statement logic isn't littered throughout the client: "if the material is a book, calculate the fine using this algorithm, otherwise if it's a movie, calculate it that way, otherwise…."
- New FineCalculators can be introduced, and existing algorithms changed, without touching virtually any other code elsewhere in the system (a great example of Bertrand Meyer's open-closed principle).
- The interface isolates the client software from any changes to the implementation details of each FineCalculator algorithm, as long as it continues to meet its contract (which is assured by unit tests).
- The client software can be unit-tested in isolation and thus not have to depend on interacting with any one specific material type. Tests for the client can substitute a test-double that implements the same abstraction solely for purposes of testing. This ability to test against a simple, in-memory construct isolates the client code being tested from dependency on a collaborating class that might be volatile, slow, or even non-existent.

Without the interface, the client is dependent on concrete details of the algorithms, which are likely to change over time. Introducing an abstraction layer, in the form of an interface, basically nets you all the positive benefits of reduced coupling.

Generalization is also Abstraction

It would be possible to name the charge() method something like CalculateChargeForBooksOverTenDaysLate(), but that has a problem of over-specification and implementation exposure. It is not an essential feature of FineCalculator that the charge is for a book (modern libraries lend a variety of materials), nor that the charge is only calculated for lateness over 10 days. A name that reveals only relevant and correct information is an abstraction. One may back into abstraction by stripping irrelevant details from names in the system.

With a name like FineCalculator a developer can know in an instant if this is a class he wishes to subclass or not. Generalization has limits, though. The simpler name "Calculator" lacks evocative value. Uncertain whether to implement its interface, a developer may create a new interface (duplication!), hack new behavior into existing code (complication!), or directly modify the caller of the existing FineCalculators with code for calculating his specific fine (duplication and coupling!)

The TDD community has been recently buzzing with the realization that code becomes more general as tests become more specific, revealing that test-driving code alone will push it to a more appropriate level of abstraction. It is still up

to the human(s) at the keyboard to change the class and method names to match.

Data Duplication vs. Abstraction

There is a very deep antipathy between duplication and abstraction.

One frequent example we've encountered is the pervasive use of a parameterized collection object. For example, the library system works with lists of holdings:

```
List<Holding> holdings = new ArrayList<Holding>();
```

Throughout the code, you'll find dozens of references to the List<Holding> type, often in signatures or method calls:

```
List<Holding> holdings = findHoldings(patron);
public List<Holding> findHoldings(Patron patron);
    // ...
}
```

This is a subtle form of duplication: We have to specify two pieces of information—the collection type and the type to which the collection is bound—in every appropriate code place. Suppose we must now associate additional characteristics with the collection of holdings as a whole, such as a date stamp to indicate when the collection was created. We can pass this date stamp around as an additional argument here and there where appropriate:

```
public void archiveHoldings(List<Holding> holdings, Date created)
```

This opens up the door to increasingly long method signatures over time, instead of helping the system to evolve gracefully. The date is really an attribute of the list of collections as a whole—yet we have no abstraction in which we could capture that information.

Prefer instead to create an abstraction that simply encapsulates the two:

```
public class HoldingSet {
    private List<Holding> holdings = new ArrayList<Holding>();
    private Date created;
    // …
}
```

This amplifies what's important—the collection of holdings—and buries the irrelevant fact that holdings are stored as a sequential list.

As you need, you can easily incorporate new behaviors into HoldingSet without having to revisit numerous method signatures throughout the application. The abstractions become richer over time instead of the parameter lists becoming more cumbersome. Abstraction drives out duplication.

The same principle applies to a loose collection of primitive parameters. Perhaps a repeating set of (latitudeHours, latitudeMinutes, latitudeSeconds, longitudeHours, longitudeMinutes, longitudeSeconds) might indicate a missing map coordinate abstraction? Do the coordinates have related methods scattered about the code?

Code Duplication vs. Abstraction

You may frequently find two-line or even single-line duplications. In the Risk game implementation we've looked at, there is a large class named Risk which

looks to control everything about the game. Within this multi-thousand-line class are numerous methods and lines of code involving both an offensive player (attacker) and a defensive player:

```
public boolean isValidAttack(Player attacker, Player defender);
if (display(attackingPlayer) || display(defendingPlayer)) {
     // ….
}
```

The class that controls the game includes additional information related to making attacks:

```
int[] attackerResults = game.rollDice(game.getAttackerDice());
int[] defenderResults = game.rollDice(game.getDefenderDice());
```

Similar code is sprinkled through the Risk class. Virtually every place there is code relating to an attacker, there is also code relating to a defender.

The related code can be rolled into a single abstraction, an *Attack*:

```
public class Attack {
    // ... fields here ...
    public Attack(Game game, Player attacker, Player defender) {
       // ...
    }
    public boolean isValid() {
        // ...
    }
    public void rollDice() {
       attackerResults = game.rollDice(game.getAttackerDice());
       defenderResults = game.rollDice(game.getDefenderDice());
    }
    public int[] getAttackerResults() {
       return attackerResults;
    }
    // ...
}
```

With this design change, you see very subtle bits of unnecessary (duplicate) code disappear. For example, we were able to change the method name isValidAttack to isValid, once we moved it into the Attack class.

The client code becomes simpler overall. We've moved two lines of complexity involving interaction with a game object into a single method in the Attack class, rollDice. That change didn't eliminate any duplication yet, but it did simplify the client and achieved command-query separation (i.e. we can ask for attacker and defender results multiple times without having to re-roll the dice):

```
Attack attack = new Attack(game, attacker, defender);
attack.rollDice();
int[] attackerResults = attack.getAttackerResults();
int[] defenderResults = attack.getDefenderResults();
```

Further, we made it possible to change the implementation of how dice are rolled without having to open and touch the client class. The game object is now referenced in the client only when constructing the Attack object. The design isn't yet "perfect"—perhaps we should move the rollDice, getAttackerDice, and getDefenderDice methods into the Attack class itself—but we now have a new home into which we can relocate attack-related code.

With the introduction of this previously missing abstraction, our many-thousand-line blob class shrinks by perhaps a few dozen lines of code. As the Risk class shrinks over time, additional opportunities for abstraction become more obvious. Abstraction begets abstraction.

Spotting "missing" abstractions takes a bit of practice. Here are a few smells that might point to the need for additional abstractions:

- Code chunks that seem to repeat (perhaps not exactly) throughout the code.
- ctrl-c / ctrl-v
- "I know I saw something similar somewhere else in the code."
- Extensive detailed test setup

Tiny Abstractions

Sometimes, you'll spot two lines, or even a single line, that redundantly specifies code. Here's a bit of ugliness used to add two new menu items, and corresponding actions, to an existing menu:

```
// clean
this.clean = new MenuItem(this.menu, SWT.PUSH);
this.clean.addSelectionListener(
    App.instance().getAction(CleanAction.NAME));
// remove
this.remove = new MenuItem(this.menu, SWT.PUSH);
this.remove.addSelectionListener(
    App.instance().getAction(RemoveAction.NAME));
```

Don't hesitate to factor these couplets into a single method! While they may not represent a top-level abstraction like a class, helper methods in the same class are still abstractions—you're replacing a complex implementation detail with a simple declaration:

```
this.clean = createMenuItem(menu, CleanAction.NAME);
this.remove = createMenuItem(menu, RemoveAction.NAME);
```

And once you've created such methods, you may start to notice that they too may be better suited in another class, whether existing or new. Further, you might recognize that things are a bit disjoint and implicit—it seems as if there's an action object somewhere that is associated to the key identified by CleanAction. A good goal for this code might be to shape it into something like:

```
this.clean = menu.addItem(new CleanAction());
this.remove = menu.addItem(new RemoveAction());
```

Of course, the library type for menu may not support this—perhaps it's time to create your own abstractions that wrap the third-party types.

We hear the same resistance to these ideas all the time: "But all these new method calls and object instantiations are going to degrade performance." When we hear this, we recommend that the programmers try and measure. You will be surprised to find what is fast, what is slow, and why. The world changes too fast to blindly follow rules of thumb about performance.

Test Abstraction

Unit tests, particularly those created as a virtue of doing test-driven development (TDD), must document the essence of what's going on:

- What data is being created for purposes of the test?
- What behavior is being executed?
- How do we know that the expected behavior happened?

It's far too easy to drown these three key test elements in a sea of difficult-to-understand test code.

Tests must amplify what's essential and bury what's not relevant to understanding the requirement. Tests that are not sufficiently abstract will be difficult to understand and will break for all the wrong reasons. Test abstraction is such a significant element of doing TDD well that we've chosen to discuss it in an upcoming article.

Conclusion

Abstraction is where object-oriented software design starts. We strive to build a system that presents straightforward concepts to the reader, not overwhelming masses of detail.

The process of abstracting drives out duplication and reveals more natural abstractions over time, making the code easier to read and easier to test.

A well-abstracted design imparts meaning and provides easy navigation. We can deftly navigate the system through its simple abstractions, and push them aside when we need to get to the nitty-gritty implementation details.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U3] with Tim, he's written another couple books, *Agile Java* and *Essential Java Style*, contributed to Uncle Bob's *Clean Code*, and written over 90 articles on software development. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

About Tim

Tim Ottinger is the other author of Agile in a Flash [U4], another contributor to *Clean Code*, a 30-year (plus) software developer, agile coach, trainer, consultant, incessant blogger, and incorrigible punster. He writes code. He likes it.

Send the authors your feedback [U5] or discuss the article in the magazine forum [U6].

External resources referenced in this article:

[U1] http://pragprog.com/magazines/2010-12/cohesive-software-design

[U2] http://pragprog.com/magazines/2011-01/code-coupling

[U3] http://www.pragprog.com/refer/pragpub20/titles/olag/Agile-in-a-flash

[U4] http://www.pragprog.com/refer/pragpub20/titles/olag/Agile-in-a-flash

[U5] mailto:michael@pragprog.com?subject=Agile-cards

[U6] http://forums.pragprog.com/forums/134