Extracted from:

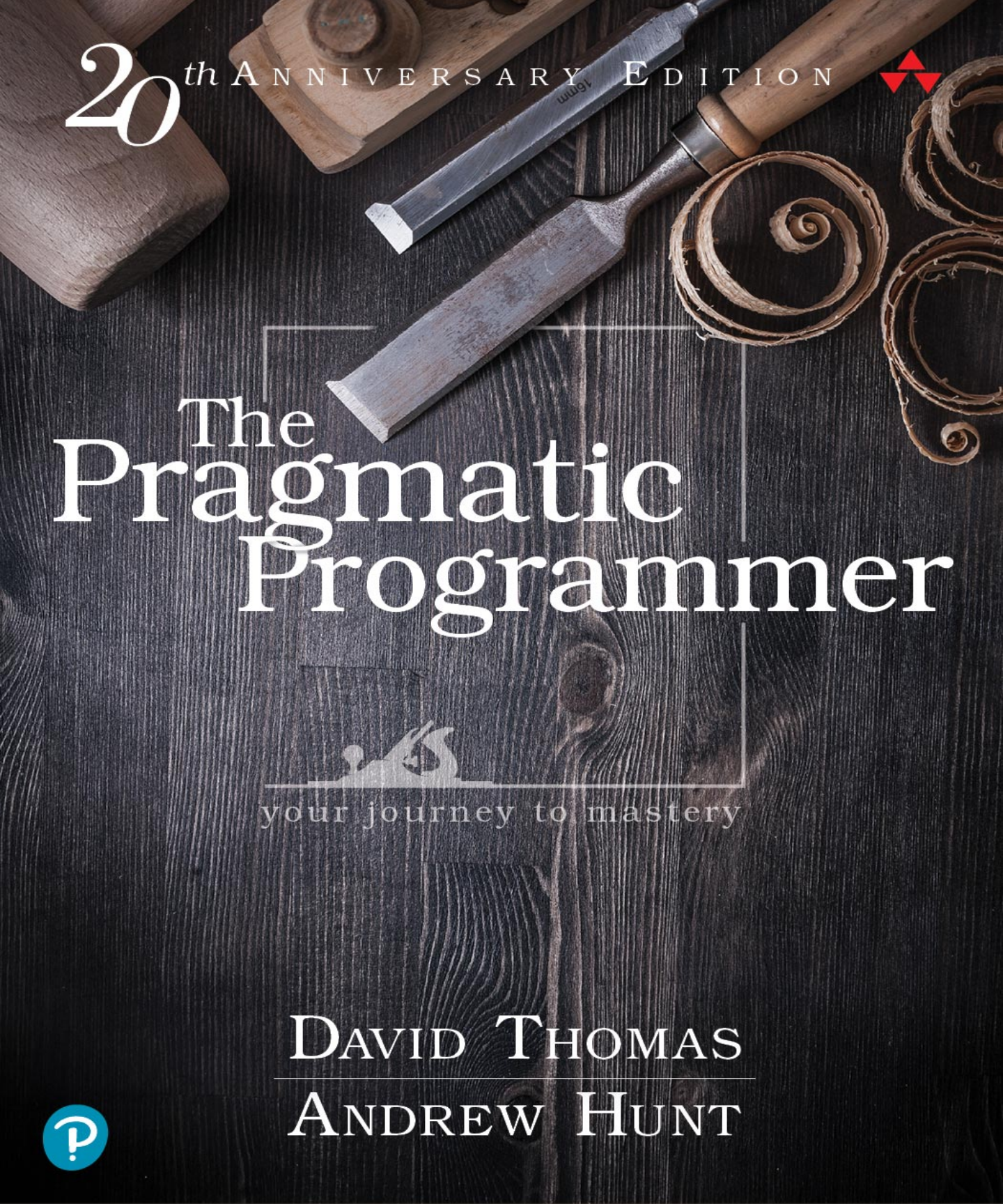
The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo


20th Anniversary Edition
The Pragmatic Programmer
your journey to mastery
David Thomas
Andrew Hunt

The Pragmatic Programmer

your journey to mastery

20th Anniversary Edition

Dave Thomas
Andy Hunt

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Library of Congress Control Number: [to come from ITP]

ISBN-13: 978-0-13-595705-9
ISBN-10: 0-13-595705-2

1 19

Inheritance Tax

You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

➤ *Joe Armstrong*

Do you program in an object-oriented language? Do you use inheritance?

If so, stop! It probably isn't what you want to do.

Let's see why.

Some Background

Inheritance first appeared in Simula 67 in 1969. It was an elegant solution to the problem of queuing multiple types of events on the same list. The Simula approach was to use something called *prefix classes*. You could write something like this:

```
link CLASS car;
  ... implementation of car

link CLASS bicycle;
  ... implementation of bicycle
```

You could then add both cars and bicycles to the list of things waiting at (say) a traffic light. In current terminology, link would be a parent class.

The mental model used by Simula programmers was that the instance data and implementation of class link was prepended to the implementation of classes car and bicycle. The link part was almost viewed as being a *container* that carried around cars and bicycles. This gave them a form of polymorphism: cars and bicycles both implemented the link interface because they both contained the link code.

After Simula came Smalltalk. Alan Kay, one of the creators of Smalltalk, describes in a 2019 Quora answer[8] *why* Smalltalk has inheritance.

> So when I designed Smalltalk-72—and it was a lark for fun while thinking about Smalltalk-71—I thought it would be fun to use its Lisp-like dynamics to do experiments with "differential programming" (meaning: various ways to accomplish "this is like that except").

8. https://www.quora.com/What-does-Alan-Kay-think-about-inheritance-in-object-oriented-programming

This is subclassing purely for behavior.

These two styles of inheritance (which actually had a fair amount in common) developed over the following decades. The Simula approach, which suggested inheritance was a way of combining types, continued in languages such as C++ and Java. The Smalltalk school, where inheritance was a dynamic organization of behaviors, was seen in languages such as Ruby and JavaScript.

So, now we're faced with a generation of OO developers who use inheritance for one of two reasons:

- they don't like typing
- they like types

Those who don't like typing save their fingers by using inheritance to add common functionality from a base class into child classes: class `User` and class `Product` are both subclasses of `ActiveRecord::Base`.

Those who like types use inheritance to express the relationship between classes: a `Car` is-a-kind-of `Vehicle`.

Unfortunately both kinds of inheritance have problems.

Problems Using Inheritance to Share Code.

Inheritance *is* coupling. Not only is the child class coupled to the parent, the parent's parent, and so on; but the code that *uses* the child is also coupled to all the ancestors.

```
class Vehicle
  def initialize
    @speed = 0
  end
  def stop
    @speed = 0
  end
  def move_at(speed)
    @speed = speed
  end
end

class Car < Vehicle
  def info
    "I'm car driving at #{@speed}"
  end
end

# top-level code
my_ride = Car.new
```

```
my_car.move_at(30)
```

When the top-level calls my_car.move_at, the method being invoked is in Vehicle, the parent of Car.

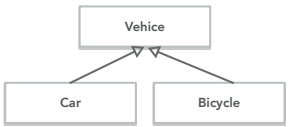
Now the developer in charge of Vehicle changes the API, so move_at becomes set_velocity, and the instance variable @speed becomes @velocity.

An API change is expected to break clients of Vehicle class. But the top-level is not: as far as it is concerned it is using a Car. What the Car class does in terms of implementation is not the concern of the top-level code, but it still breaks.

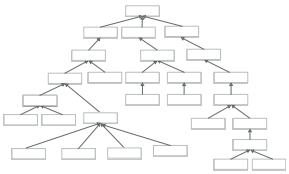
Similarly the name of an instance variable is purely an internal implementation detail, but when Vehicle changes it also (silently) breaks Car.

So much coupling.

Problems Using Inheritance To Build Types



Some folks view inheritance as a way of defining new types. Their favorite design diagram shows class hierarchies. They view problems the way Victorian gentleman scientists viewed nature, as something to be broken down into categories.



Unfortunately, these diagrams soon grow into wall-covering monstrosities, layer-upon-layer added in order to express the smallest nuance of differentiation between classes. This added complexity can make the application more brittle, as changes can ripple up and down many layers.

Even worse, though, is the multiple inheritance issue. A Car may be a kind of Vehicle, but it can also be a kind of Asset, InsuredItem, LoanCollateral and so on. Modeling this correctly would need multiple inheritance.

C++ gave multiple inheritance a bad name in the 1990s because of some questionable disambiguation semantics. As a result, many current OO languages don't offer it. So, even if you're happy with complex type trees, you won't be able to model your domain accurately anyway.

Tip 51 Don't Pay Inheritance Tax

The Alternatives Are Better

Let us suggest three techniques that mean you should never need to use inheritance again:

- interfaces and protocols
- delegation
- mixins and traits

Interfaces and Protocols

Most OO languages allow you to specify that a class implements one or more sets of behaviors. You could say, for example, that a Car class implements the Drivable behavior and the Locatable behavior. The syntax used for doing this varies: in Java, it might look like this:

```
public class Car implements Drivable, Locatable {
  // ...
}
```

Drivable and Locatable are what Java calls *interfaces*; other languages call them *protocols*, and some call them *traits* (although this is not what we'll be calling a trait later).

Interfaces are defined like this:

```
public interface Drivable {
  double getSpeed();
  void   stop();
}

public interface Locatable() {
  Coordinate getLocation();
  boolean    locationIsValid();
}
```

These declarations create no code: they simply say that any class that implements Drivable must implement the two methods getSpeed and stop, and a class that's Locatable must implement getLocation and locationIsValid. This means that our previous class definition of Car will only be valid if it includes all four of these methods.

What makes interfaces and protocols so powerful is that we can use them as types, and any class that implements the appropriate interface will be compatible with that type. If Car and Phone both implement Locatable, we could store both in an list of locatable items:

```
List<Locatable> items = new ArrayList<>();
```

```
items.add(new Car(...));
items.add(new Phone(...));
items.add(new Car(...));
// ...
```

We can then process that list, safe in the knowledge that every item has `getLocation` and `locationIsValid`.

```
void printLocation(Locatable item) {
  if (item.locationIsValid() {
    print(item.getLocation().asString());
}

// ...

items.forEach(printLocation);
```

Tip 52 Prefer Interfaces To Express Polymorphism

Interfaces and protocols give us polymorphism without inheritance.

Delegation

Inheritance encourages developers to create classes whose objects have large numbers of methods. If a parent class has 20 methods, and the subclass wants to make use of just two of them, its objects will still have the other 18 just lying around and callable. The class has lost control of its interface. This is a common problem: many persistence and UI frameworks insist that application components subclass some supplied base class:

```
class Account < PersistenceBaseClass
end
```

The `Account` class now carries all of the persistence class's API around with it. Instead, imagine an alternative using delegation:

```
class Account
  def initialize(. . .)
    @repo = Persister.for(self)
  end

  def save
    @repo.save()
  end
end
```

We now expose *none* of the framework API to the clients of our `Account` class: that decoupling is now broken. But there's more. Now that we're no longer constrained by the API of the framework we're using, we're free to create the

API we need. Yes, we could do that before, but we always ran the risk that the interface *we* wrote can by bypassed, and the persistence API used instead. Now we control everything.

Tip 53 Delegate to Services: Has-A Trumps Is-A

In fact, we can take this a step further. Why should an `Account` have to know how to persist itself? Isn't its job to know and enforce the account business rules?

```
class Account
  # nothing but account stuff
end

class AccountRecord
  # wraps an account with the ability
  # to be fetched and stored
end
```

Now we're really decoupled, but it has come at a cost. We're having to write more code, and typically some of it will be boilerplate: it's likely that all our record classes will need to `find` method, for example.

Fortunately, that's what mixins and traits do for us.

Mixins, Traits, Categories, Protocol Extensions, …

As an industry, we love to give things names. Quite often we'll give the same thing many names. More is better, right?

That's what we're dealing with when we look at mixins. The basic idea is simple: we want to be able to extend classes and objects with new functionality without using inheritance. So we create a set of these functions, give that set a name, and then somehow extend a class or object with them. At that point, you've created a new class or object that combines the capabilities of the original and all its mixins. In most cases, you'll be able to make this extension even if you don't have access to the source code of the class you're extending.

Now the implementation and name of this feature varies between languages. We'll tend to call them *mixins* here, but we really want you to think of this as a language-agnostic feature. And we'll focus on just the functional that all these implementations have: merging functionality between existing *things* and new *things*.

As an example, let's go back to our AccountRecord example. As we left it, an AccountRecord needed to know about both accounts and about our persistence framework. It also needed to delegate all the methods in the persistence layer that it wanted to expose to the outside world.

Mixins give us an alternative. First, we could write a mixin that implements (for example) two of three of the standard finder methods. We could then add them into AccountRecord as a mixin. And, as we write new classes for persisted things, we can add the mixin to them, too.

```
mixin CommonFinders {
  def find(id) { ... }
  def findAll() { ... }
end

class AccountRecord extends BasicRecord with CommonFinders
class OrderRecord   extends BasicRecord with CommonFinders
```

We can take this a lot further. For example, we all know our business objects need validation code to prevent bad data from infiltrating our calculations. But exactly what do we mean by *validation*?

If we take an account, for example, there are probably many different layers of validation that could be applied:

- validating a hashed password matches one entered by the user
- validating form data entered by the user when an account is created
- validating form data entered by an admin person updating the user details
- validating data added to the account by other system components
- validating data for consistency before it is persisted

A common (and we believe less-than-ideal) approach is to bundle all the validations into a single class (the business object/persistence object) and then add flags to control which fire in which circumstances.

We think a better way is to use mixins to create specialized classes for appropriate situations:

```
class AccountForCustomer extends Account
     with AccountValidations,AccountCustomerValidations

class AccountForAdmin extends Account
     with AccountValidations,AccountAdminValidations
```

Here, both derived classes include validations common to all account objects. The customer variant also includes validations appropriate for the customer-

facing APIs, while the admin variant contained (the presumably less restrictive) admin validations.

Now, by passing instances of `AccountForCustomer` or `AccountForAdmin` back and forth, our code *automatically* ensures the correct validation is applied.

Tip 54 Use Mixins to Share Functionality

Inheritance is Rarely the Answer

We've had a quick look at three alternatives to traditional class inheritance:

- interfaces and protocols
- delegation
- mixins and traits

Each of these methods may be better for you in different circumstances, depending on whether your goal is sharing type information, adding functionality, or sharing methods. As with anything in programming, aim to use the technique that best expresses your intent.

And try not to drag the whole jungle along for the ride.

Related Sections Include

- Topic 28, *Decoupling*, on page ?
- Topic 8, *The Essence of Good Design*, on page ?
- Topic 10, *Orthogonality*, on page ?

Challenges

- The next time you find yourself subclassing, take a minute to examine the options. Can you achieves what you want with interfaces, delegation, and/or mixins? Can you reduce coupling by doing so?