Implementing Object-oriented Designs

Alex Korban
AUTHOR, DEVELOPER
@alexkorban korban.net

Overview

Apply language features in alignment with OOP principles

Write cohesive classes and modules

Achieve loose coupling

Adhere to the Liskov substitution principle

Understand tradeoffs between inheritance, mixins, and composition

Delegate method calls with *Forwardable*

Principles of Object-oriented Design

Cohesive Classes and Modules

Loose Coupling

Liskov Substitution Principle

Don’t Repeat Yourself

Each class and module should have a single responsibility. Aim for smaller classes and modules with a specific purpose.

Cohesive Classes and Modules

```
class User
  def error(msg)
    @logger.log(“[ERROR]", msg)
  end

  def info(msg)
    @logger.log(“[INFO]", msg)
  end

  def email_announcement(new_releases)
    # ...
  end
end
```

Cohesive Classes and Modules

```
module Log
 @logger = Logger.new

  def self.error(msg)
    @logger.log(“[ERROR]", msg)
  end

  def self.info(msg)
    @logger.log(“[INFO]", msg)
  end
end
```

Cohesive Classes and Modules

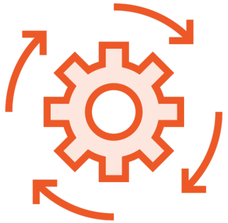
```
module Mailer
  def send_announcement(user, books)
    # ...
  end

  def send_broadcast(users)
    # ...
  end
end
```

Cohesive Classes and Modules

```
class User
  def send_announcement(books)
    @mailer.send_announcement(self, books)
  end
end
```

Loose Coupling

Replace components easily as requirements change



Facilitate refactoring and testing

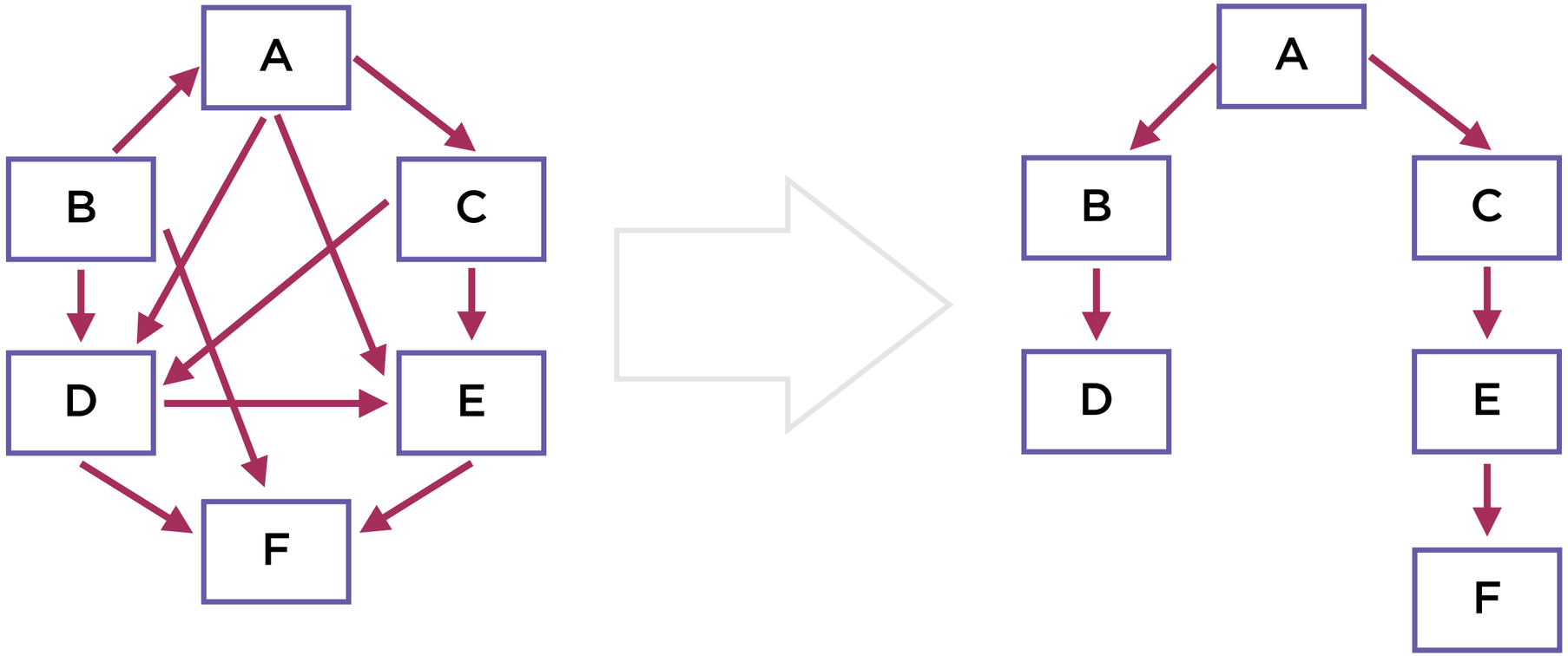
Types of Coupling

Classes are coupled via inheritance

Classes including or extending modules

References to classes, methods, constants

Loose Coupling



Duck Typing

```
exporter = if export_format == :csv
  CSVExporter.new(current_user)
else
  JSONExporter.new(current_user)
end

exporter.export(filename)
```

Dependency Injection

Can help reduce coupling

Instantiating objects inside a class creates dependencies

You can pass objects in instead

Easier to substitute objects of different classes

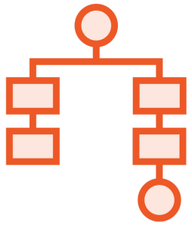
Makes classes more testable

Dependency Injection

```ruby
class User
  def initialize(id)
    @id = id
    @mailer = Mailer.new
  end
end
```

```ruby
class User
  def initialize(id, mailer = Mailer.new)
    @id = id
    @mailer = mailer
  end
end
```

Implementation Details



Implementation details are exposed by default

Better to expose only the methods that form a useful interface

Avoid providing public methods that could put object in an invalid state

```
class User
  def add_card(card)
  end

  def delete_card!
  end
end
```

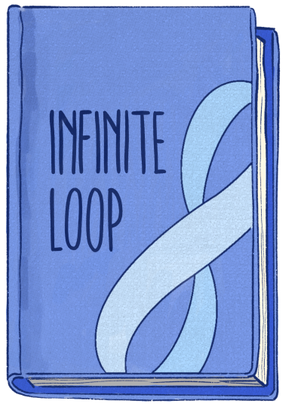
Handling Billing in the User Class

A user shouldn't be able to update but not remove credit card details

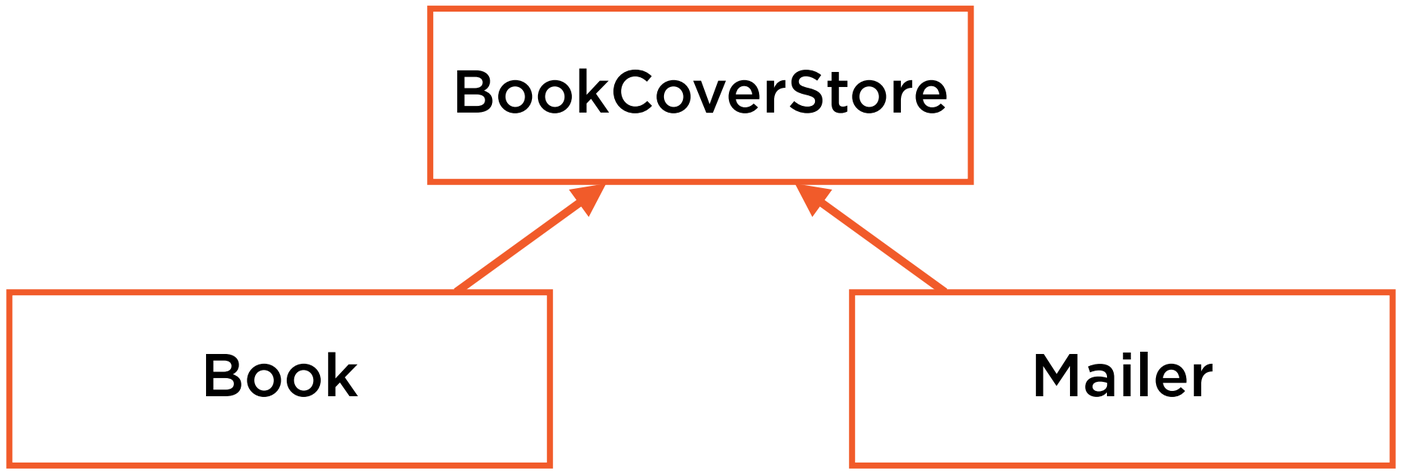
Handling Billing in the User Class

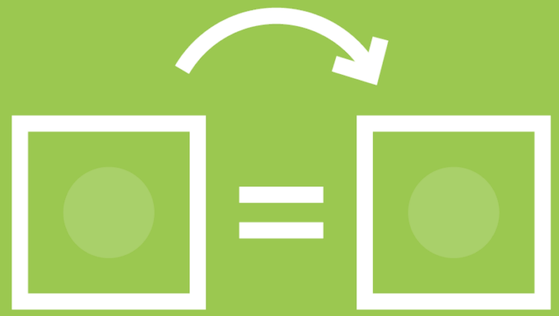
```
class User
  def set_card(card)
    delete_card!() if card_exists?
    add_card(card)
  end

  private

  def add_card(card)
  end

  def delete_card!
  end
end
```

Implicit Coupling



/public/img/covers/123-infinite-loop-300x300.png

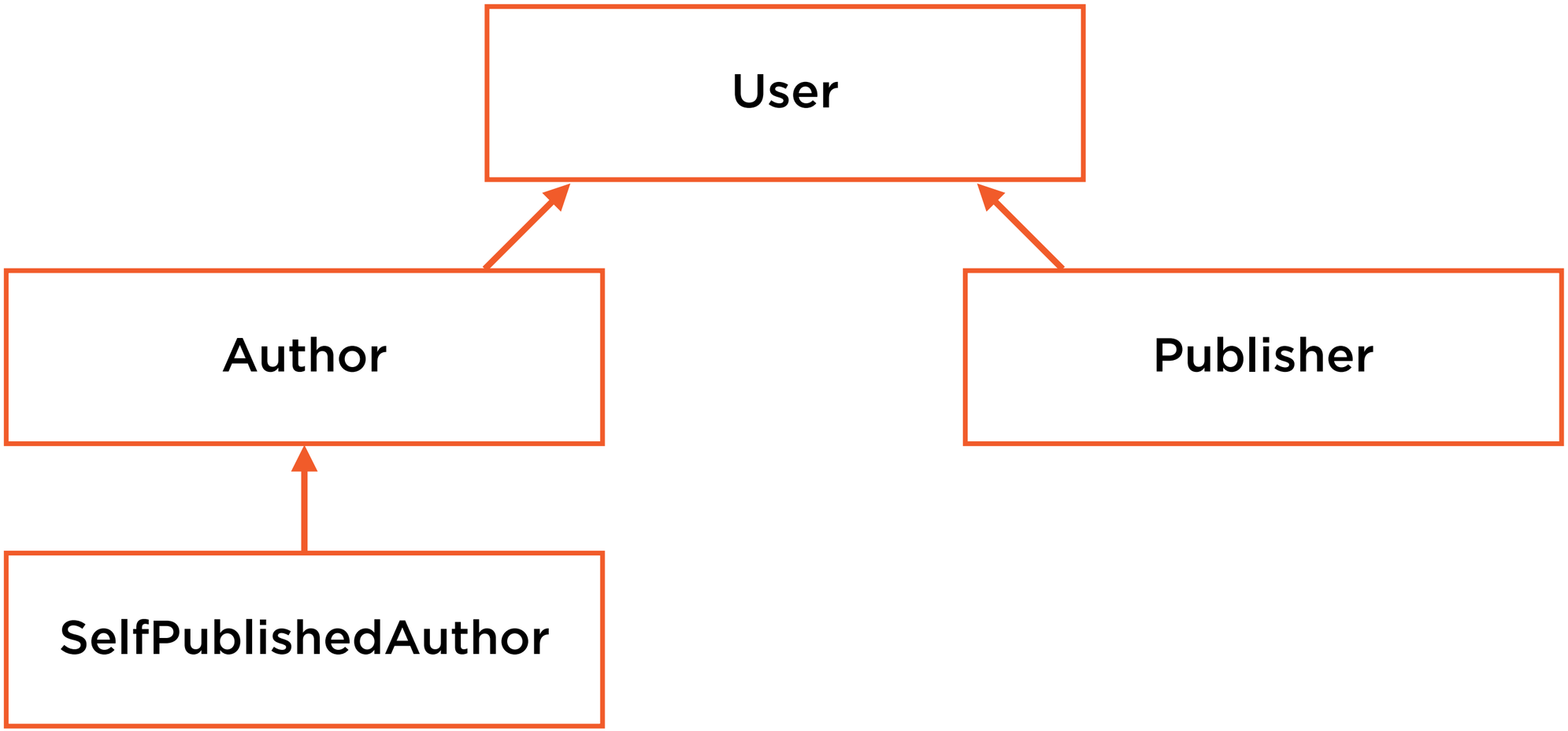




Liskov Substitution Principle

You should be able to use an instance of a subclass wherever an instance of a superclass is expected

Liskov Substitution Principle



Handling Billing in the User Class

```ruby
class User
  def charge!
  end

  def payment_due?
  end
end
```

```ruby
class Author < User
  def charge!
    # Do something else
  end

  def bill!
  end
end
```

```ruby
users.each {|user| user.charge! if user.payment_due? }
```

Don’t Repeat Yourself

Not specific to object-oriented programming

Minimize code duplication by abstracting common parts

Mixins and inheritance facilitate this

You should have a single source of truth for every bit of knowledge in your application

```
class User
  def initialize(name)
    @name = name
    @mailer = Mailer.new(@name)
  end
end
```

Single Source of Truth

```
class User
  def initialize
    @mailer = Mailer.new
  end

  def send_announcement
    @mailer.send_email(@name, content)
  end
end
```

Single Source of Truth

Single Source of Truth

Ensure that classes encapsulate non-overlapping sets of data

Compute derived bits of data instead of storing results

Apply caching judiciously

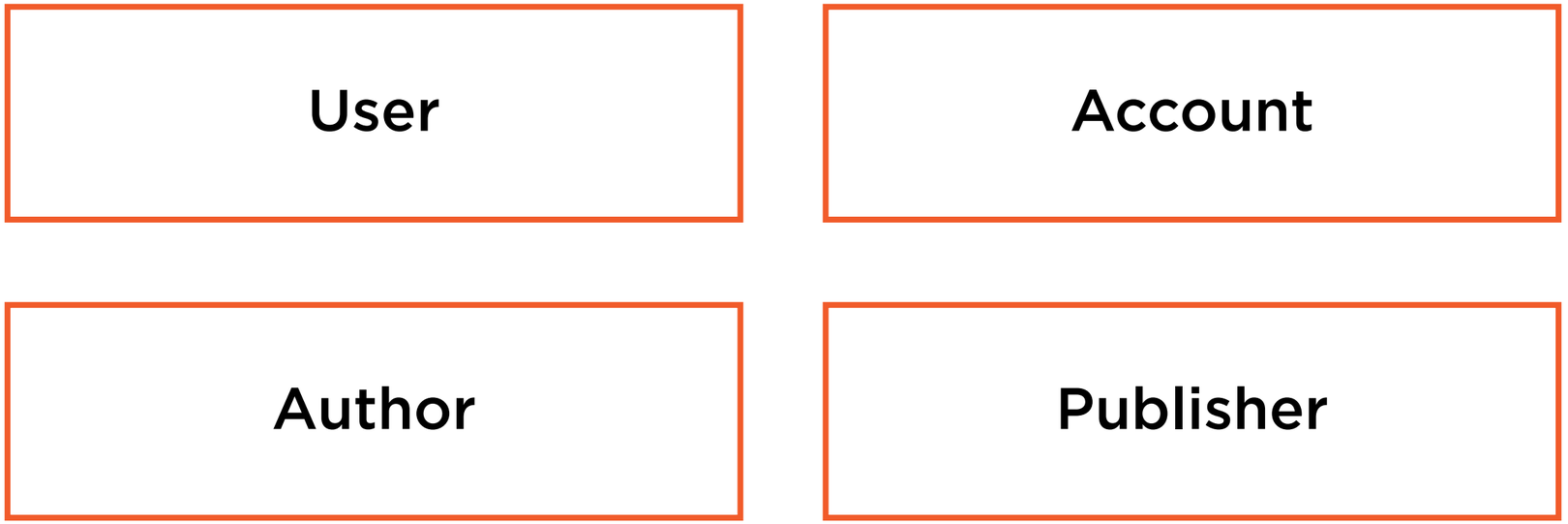
Structuring the Code

Structuring the Code

Prefer composition to inheritance

Prefer mixins to inheritance

Mixins



Mixins

```ruby
module Account
end

module UserFunctions
end

module AuthorFunctions
end

module PublisherFunctions
end
```

```ruby
class User
  include Account
  include UserFunctions
end

class Author
  include Account
  include UserFunctions
  include AuthorFunctions
end
```

```
class SelfPublishedAuthor
  include Account
  include UserFunctions
  include AuthorFunctions
  include PublisherFunctions
end
```

Mixins

Composition

```ruby
class Account
  def charge!
  end
end

class User
  def initialize(id)
    @account = Account.new(id)
  end
end

class Author
  def initialize(id)
    @user = User.new(id)
  end
end
```

```ruby
author.user.account.charge!
```

Mixins

```
class Account
  def charge!
  end
end

class Mailer
  def send_email
  end
end
```

```
class User
  def initialize(id)
    @account = Account.new(id)
    @mailer = Mailer.new
  end
end

class Author < User
end
```

Forwardable allows you to make an object delegate method calls to other objects

Delegation

Forwarding method calls allows to build an ergonomic interface

Supplant some uses of inheritance

Selectively make some methods of a member available on an object

Inheritance

Classes follow Liskov substitution principle

Models *is-a* relationships

Subclasses tweak behaviour of superclasses

Features to use with caution

Some features can lead to confusing code if overused

```ruby
class Collection
  def <<(book)
    @books << book
  end
end

collection << Book.new(title: "Code", author: "Ruby Red")
```

Custom Operators

Custom Operators

Acceptable if they mimic well known semantics

Regular methods are preferable most of the time

Operator symbols provide less insight than method names

method_missing

```
collection.find_by_name

collection.find_by_author

collection.find_by_author_and_year
```

```
collection.find(name: "Ruby Red")
```

Monkey Patching and Refinements

May cause behavior other developers don’t expect

May make code more brittle with regard to gem updates

Consider composition and delegation first

In This Course

Summary

- Striking a balance between different OOP features
- Classes and modules with a single responsibility
- Loose coupling
- Tradeoffs between inheritance, mixins, and composition
- Delegation with *Forwardable*