

From Design Principles to SOLID

Core elements of good (object-oriented) design –

- properties
 - cohesion – relatedness within a class
 - coupling – interconnections between classes
- design principles/goals
 - separation of concerns – pushes towards high cohesion
 - information hiding – pushes towards loose coupling
- mechanisms for achieving those goals
 - encapsulation – the language-level tools for information hiding
 - code to the interface
 - Law of Demeter

From Design Principles to SOLID

- *SOLID* is a set of OOD principles intended to make code more understandable, flexible, and maintainable
 - SRP: Single Responsibility Principle
 - OCP: Open-Closed Principle
 - LSP: Liskov Substitution Principle
 - ISP: Interface Segregation Principle
 - DIP: Dependency Inversion Principle
- a related heuristic
 - prefer composition to inheritance



due to Robert C.
Martin (Uncle Bob)

Single Responsibility Principle (SRP)

The *Single Responsibility Principle* (SRP) states that

A class should have one, and only one, reason to change.

...which means that a class should only be responsible for a single piece of functionality – i.e. have a single responsibility

- the more things a class is responsible for handling, the more reasons there are that the class might need to change should something in the program requirements change
 - changing one class may require also updating anything that depends on it, so limiting the scope of change is important
- fix: split by responsibility

SRP

```
/**
 * Directory information for a person.
 */
public class Person {
    private String name_;
    private String address_;

    public Person ( String name, String address ) { ... }

    public String getName () { ... }
    public String getAddress () { ... }

    private boolean validateAddress () { ... }
}
```

- violates SRP
 - two reasons to change: the “business logic” of the person itself, and how to validate an address
 - fix by putting address validation in a separate class

SRP

```
/**
 * A playing card from a standard deck.
 */
public class Card {
    private Suit suit_;
    private int value_;
    private boolean faceup_;

    public Card ( Suit suit, int value ) { ... }

    public Suit getSuit () { ... }
    public int getValue () { ... }
    public boolean isFaceUp () { ... }

    public void setFaceUp ( boolean faceup ) { ... }

    public void draw ( GraphicsContext2D g ) { ... }
}
```

- things that draw/print/save themselves violate SRP
 - two reasons to change: the “business logic” of the class itself, and how to draw/print/save
 - fix by putting draw/print/save in a separate class

SRP

```
/**
 * A library book.
 */
public class Book {
    private String title_, author_;
    private String content_;
    private int curpage_;
    private String location_;

    public Book ( String title, String author, String content ) { ... }

    public String getTitle () { ... }
    public String getAuthor () { ... }
    public String getContent () { ... }
    public String getPage ( int page ) { ... } // get specified page

    public String getPage () { ... } // get current page
    public void turnPage () { ... } // advance to next page

    public void getLocation () { ... } // book's location in the library
}
```

- violates SRP
 - three reasons to change: the “business logic” of the book itself, how the book’s content is accessed (page-by-page), how the library is organized
 - can also think in terms of user interests
 - both library patron and librarian are interested in a book’s title, author
 - library patron is interested in the book’s content and how it is accessed
 - librarian controls the book’s location

Open-Closed Principle

Classes should be open for extension, but not modification.

- it should be possible to add new behavior without editing existing, already-tested code

Why?

- flexibility demands that you be able to change things
- changing existing (and debugged!) code runs the risk of introducing bugs or breaking dependencies
- you don't always have access to the source code

You need to be able to find all of the thingamajigs in the bin, so you write some code...

```
public class ThingFilter {
    public List<Thing> filter ( List<Thing> things, String kind ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( kind.equals("Thingamajig") && thing instanceof Thingamajig ) {
                matches.add(thing);
            }
        }
        return matches;
    }
}
```

There may be a need to also find widgets and whatchamacallits in the future.

True/false: This design satisfies the Open-Closed Principle.

True

False

adding new kinds of things requires adding another case to the if

You need to be able to find all of the things with a particular color, so you write some code...

```
public class ThingFilter {
    public List<Thing> filter ( List<Thing> things, Color color ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( thing.getColor() == color ) {
                matches.add(thing);
            }
        }
        return matches;
    }
}
```

There might be a need to filter by other properties (size, kind of thing, ...) in the future.

True/false: This design satisfies the Open-Closed Principle.

True
False

adding new properties requires adding another method with the appropriate parameter, plus changing the if

Designing for OCP

How can you modify the behavior of a class without modifying its source code?

- subclass

Strategy.

- avoid 'if' statements based on the kind of thing
- instead, *encapsulate what varies* so that things that may change are handled in subclasses

Tips for designing for OCP.

- be careful about the use of `private` and `final` because those things are impossible to modify in a subclass

Encapsulate What Varies

Encapsulate what varies refers to separating elements that can vary from those that don't.

Why?

- avoid re-implementing/copying-and-pasting/re-debugging/etc common elements in multiple places
- hide potential variation, so if (when) there are changes, the rest of the program isn't affected
- allow simultaneous support of multiple variations
- allow runtime selection of which variation

Often achieved by pulling common elements into an interface or superclass, isolating differences in separate subclasses.

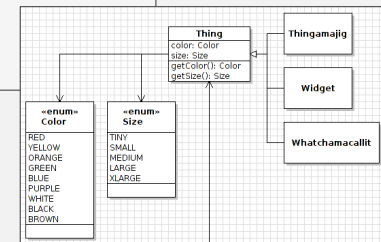
```
public class ThingFilter {
    public List<Thing> filter ( List<Thing> things, String kind ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( kind.equals( "Thingamajig" ) && thing instanceof Thingamajig ) {
                matches.add(thing);
            }
        }
        return matches;
    }
}
```

to also be able to find widgets and whatchamacallits, what varies?

– a value (the kind) which is different for different objects

→ add a `getKind` method to `Thing`

```
public class ThingFilter {
    public List<Thing> filter ( List<Thing> things, String kind ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( kind.equals(thing.getKind()) ) {
                matches.add(thing);
            }
        }
        return matches;
    }
}
```



```
public abstract class ThingFilter {
    public List<Thing> filter ( List<Thing> things ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( isMatch(thing) ) {
                matches.add(thing);
            }
        }
        return matches;
    }
    protected abstract boolean isMatch ( Thing thing )
}
```

→ encapsulate these parts in a subclass

CPSC 329: Software Development • Fall 2026

```
public class SizeFilter extends ThingFilter {
    private Size size_;
    public SizeFilter ( Size size ) { size_ = size; }
    protected boolean isMatch ( Thing thing ) {
        return thing.getSize() == size_;
    }
}
```

CPSC 329: Software Development • Fall 2026

```
public class OrFilter extends ThingFilter {
    private List<ThingFilter> filters_;
    public OrFilter () { filters_ = new ArrayList<ThingFilter>(); }
    public void addFilter ( ThingFilter filter ) { filters_.add(filter); }
    protected boolean isMatch ( Thing thing ) {
        for ( ThingFilter filter : filters_ ) {
            if ( filter.isMatch(thing) ) { return true; }
        }
        return false;
    }
}
```

CPSC 329: Software Development • Fall 2026

68



69

Questions

How does pulling common elements into an interface or superclass isolate differences in separate subclasses?

```
public abstract class ThingFilter {
    public List<Thing> filter ( List<Thing> things ) {
        List<Thing> matches = new ArrayList<Thing>();
        for ( Thing thing : things ) {
            if ( isMatch(thing) ) {
                matches.add(thing);
            }
        }
        return matches;
    }
    protected abstract boolean isMatch ( Thing thing );
}
```

nothing here depends on
the different ways to filter

```
public class ColorFilter extends ThingFilter {
    private Color color_;
    public ColorFilter ( Color color ) { color_ = color; }
    protected boolean isMatch ( Thing thing ) {
        return thing.getColor() == color_;
    }
}
```

everything about specific ways
to filter is in the subclasses

```
public class SizeFilter extends ThingFilter {
    private Size size_;
    public SizeFilter ( Size size ) { size_ = size; }
    protected boolean isMatch ( Thing thing ) {
        return thing.getSize() == size_;
    }
}
```