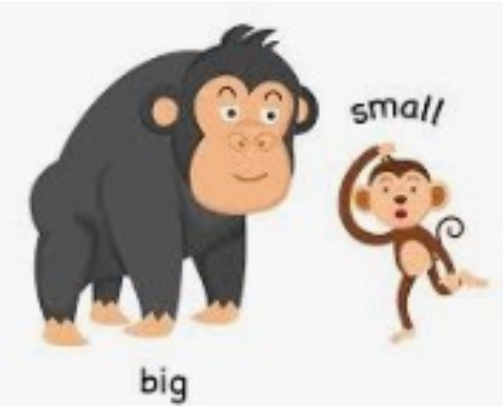
Identify areas likely to change

- Separate items likely to change from more-stable items
- Isolate the change

- Typical areas of change
  - Business rules
  - Hardware dependencies
  - Input and output
  - Non-standard language features
  - Tricky design or algorithm areas
  - Status variables

Anticipate different degrees of change

- Design so that something perceived as a small change should have a small scope of impact
  - Don’t let a small change become the 100 pound gorilla on your back



Keep coupling loose

- …said several times before in the class.
  - Nothing new to add.

Look for common design patterns

- Transformations or actions that you do repeatedly should be collected together
  - Would likely lead to refactoring later if you didn't
  - Seek common "well known" solutions
    - Boilerplate solution
    - Company standard on how to address the problem
    - Industry best practice "design pattern" solution
    - Solution already encapsulated in a library

Design Considerations

- Aim for high cohesion
- Build hierarchies
- Formalize class contracts
- Assign responsibilities
- Design for testing
- Choose the binding time consciously
- Make central points of control
- Keep your design modular

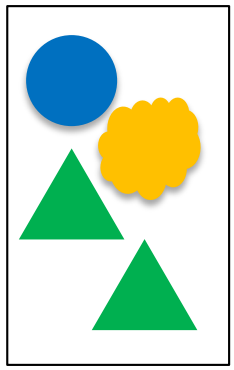
Design practices

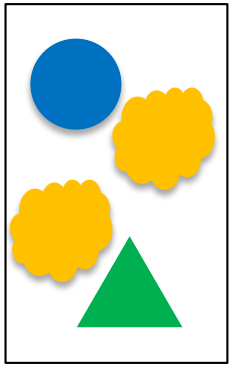
- Iterate, iterate, iterate
- Divide and conquer
- Top-down and bottom-up design
- Experimental prototyping
- Collaborative design

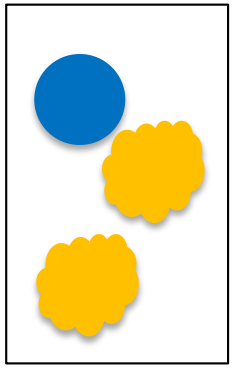
Common Design Criteria -- Cohesion

- Cohesion is a measure of relatedness to a single idea or responsibility within a method, class, or package
  - A measure of how well everything stick together
  - Aim for high cohesion
- Low cohesion means that either
  - You need to look to many methods, classes, or packages to get a task done because the pieces are fragmented or
  - One method, class, or package is trying to do a lot of different things, which makes the code difficult to understand

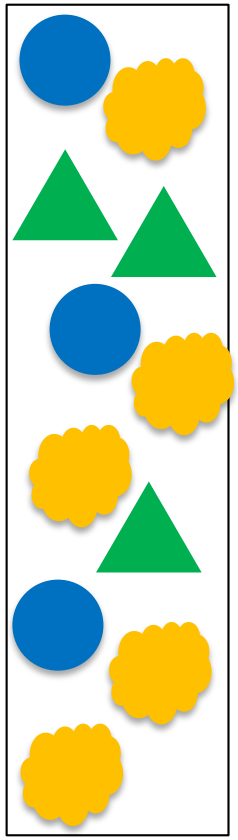
Cohesion



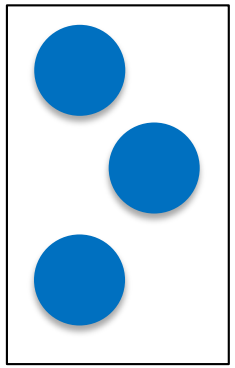


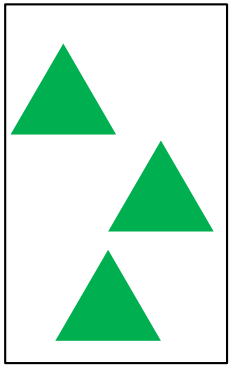


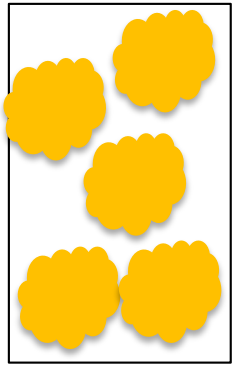
Bad cohesion
Too fragmented



Bad cohesion
Overloaded







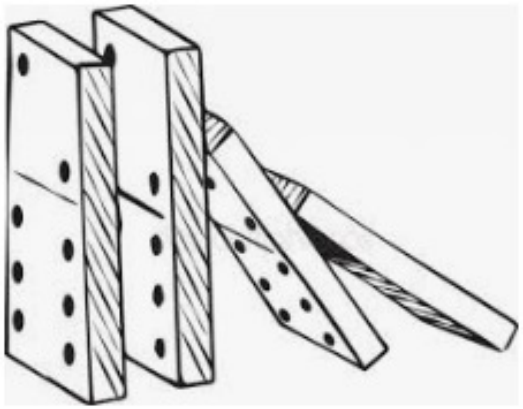
Good cohesion

DALHOUSIE UNIVERSITY
Inspiring Minds

Common Design Criteria – Coupling

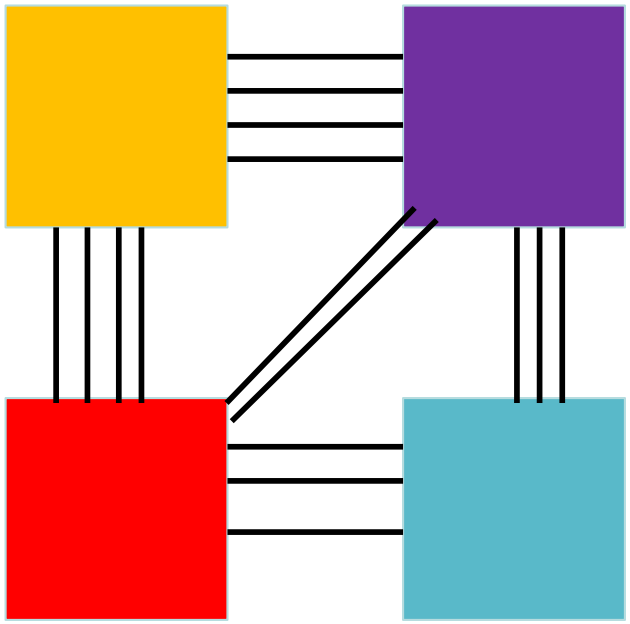
- Coupling is a measure of dependence between classes or between packages
- Aim for low or loose coupling

- High coupling makes your code difficult to change because of the ripple effect of changes that must be carried through to all the tightly coupled modules

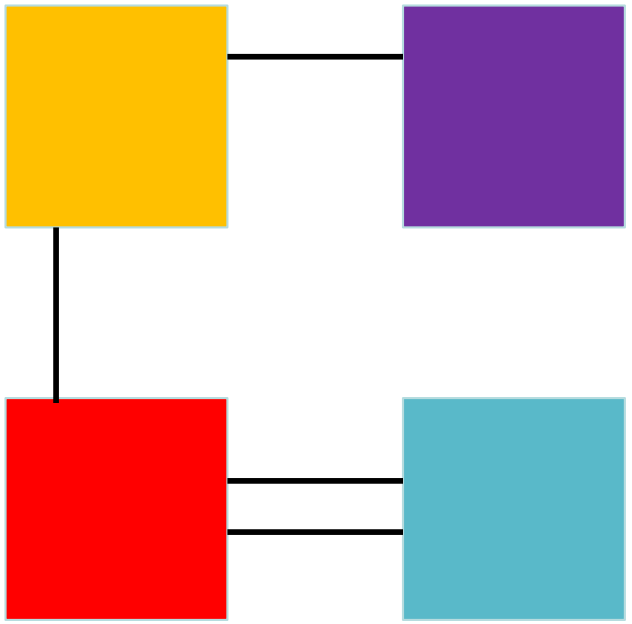


DALHOUSIE UNIVERSITY
Inspiring Minds

Coupling



High Coupling /
Tight Coupling



Low Coupling /
Loose Coupling

DALHOUSIE UNIVERSITY
Inspiring Minds

Cohesion vs. Coupling

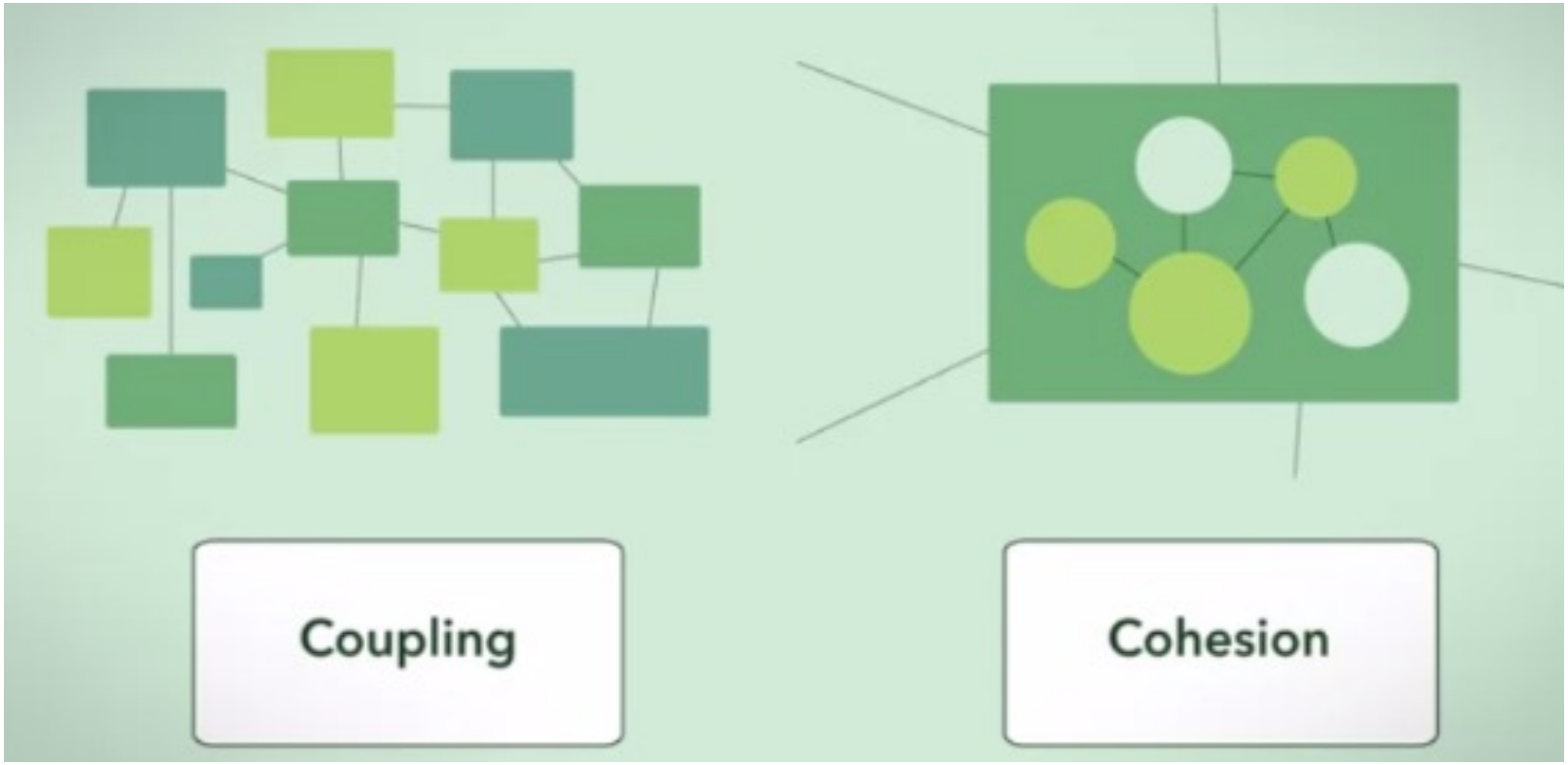


Image from StackOverflow, attributing it to
https://www.coursera.org/lecture/object-oriented-design/1-3-1-coupling-and-cohesion-q8wGt

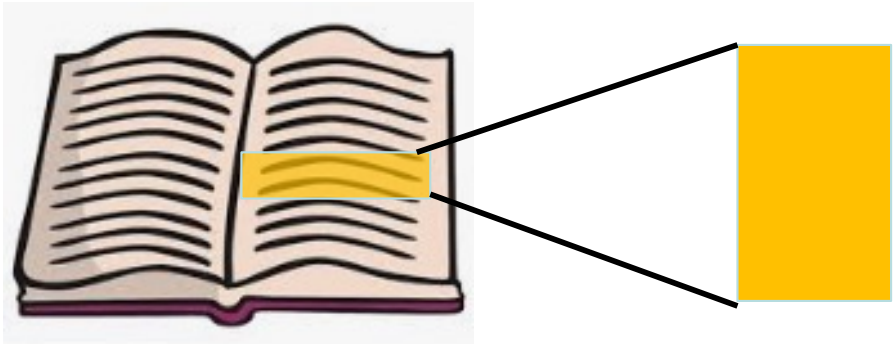
Design Principles – SOLID

- Single responsibility principle
- Open / closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

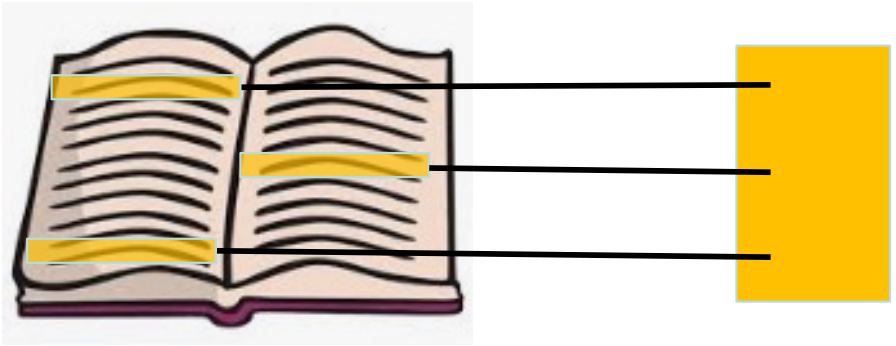
Single Responsibility Principle

- "A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class." (https://en.wikipedia.org/wiki/SOLID)

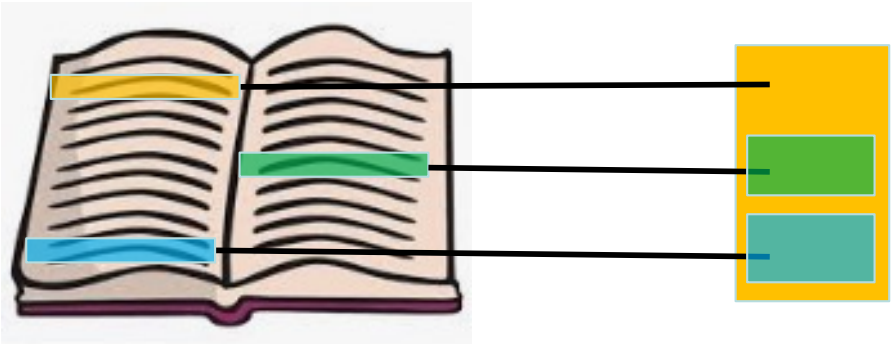
Single Responsibility Principle



Principle kept



Principle not kept



Principle really not kept

DALHOUSIE UNIVERSITY
Inspiring Minds

Single Responsibility Principle – bad example

- Public class student {
  public String getName();
  public int getAge();
  public int getFeesOwing();
  public int[ ] getRegisteredCourses();
  public boolean hasCheckedOutLibraryBook();
  }

Single Responsibility Principle – correct use

```
Public class studentInfo {
        public String getName();
        public int getAge();
}
pubic class studentFinances {
        public int getFeesOwing();
}
public class studentRegistration {
        public int[ ] getRegisteredCourses();
}
public class studentLibrary {
        public boolean hasCheckedOutLibraryBook();
}
```

Single Responsibility Principle – correct use

- Public class student {

```
        private studentInfo info;
        private studentFinances finances;
        private studentRegistration registrations;
        private studentLibrary libraryUse;
}
```

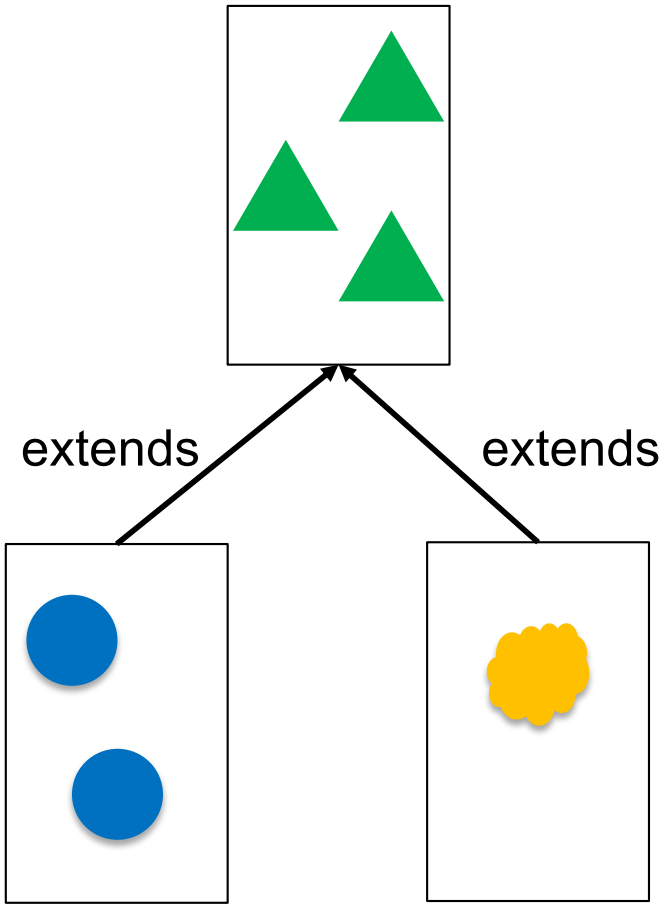
- Each smaller class has a more direct responsibility
- One aggregating class, if needed, to gather all the information
  - Don't replicate the methods of the attributes to the "student" class
  - Allow others to get references to the specific objects instead

Open / Closed Principle

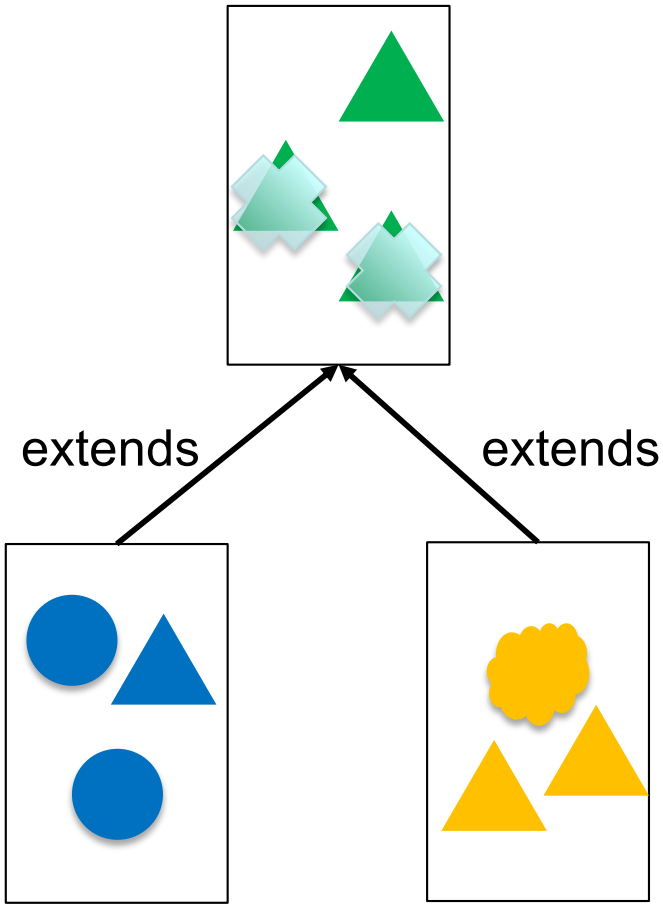
- "' Software entities ... should be open for extension, but closed for modification.'" (https://en.wikipedia.org/wiki/SOLID)

- **Relates strongly to subclasses and inheritance:**
  - **Write classes expecting / hoping that others will extend it**
    - **Better alternative than many others modifying your class**
    - **Once the class is written, we hope to not change it much**
  - **Subclasses should add functionality rather than rewrite methods from the superclass**
    - **If you need to rewrite many methods then maybe you shouldn't be extending the class**

Open

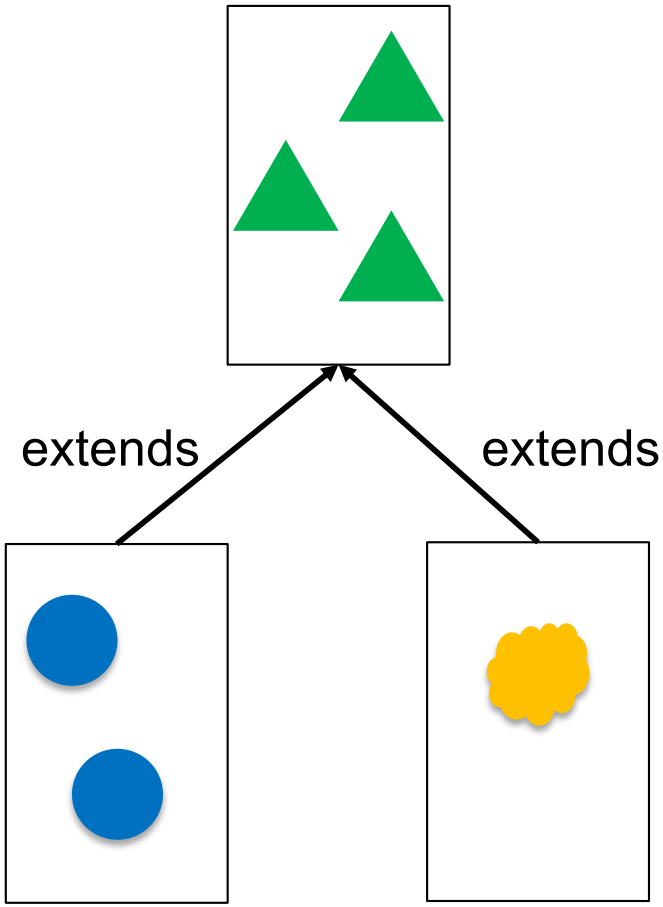


Good and expected

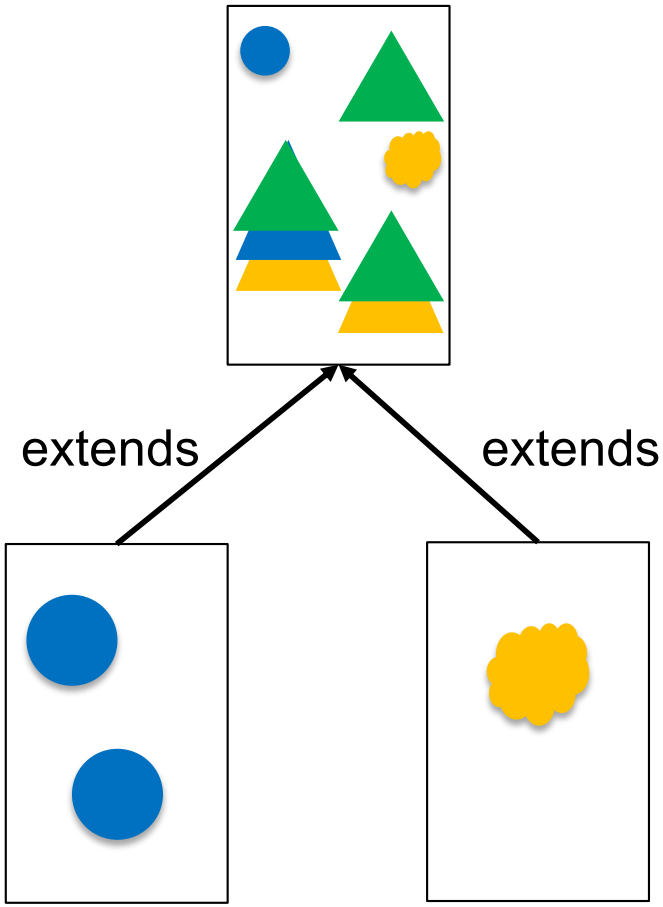


Bad since little of the parent is left

Closed

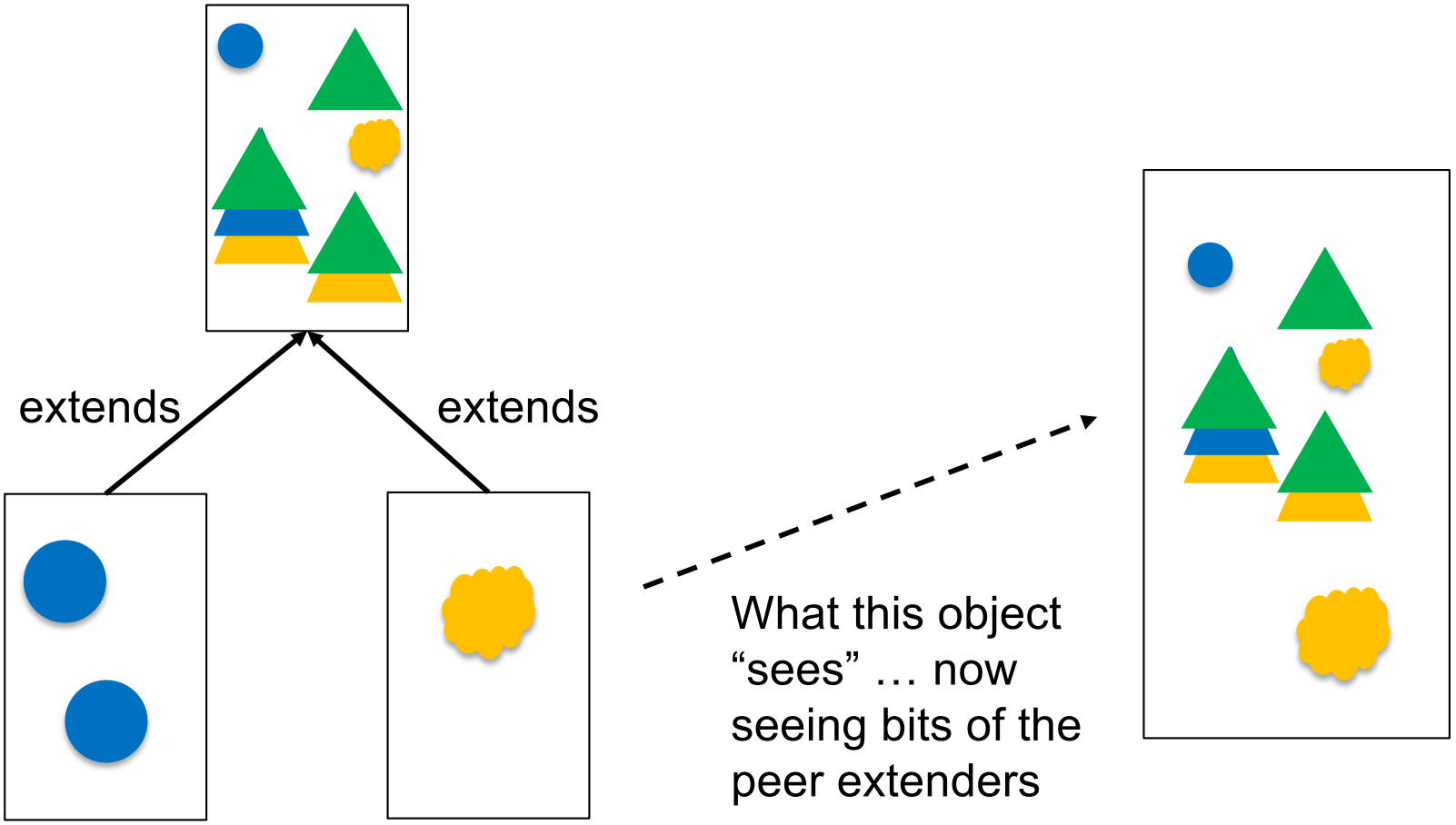


Good and expected



Bad since the parent shouldn't change

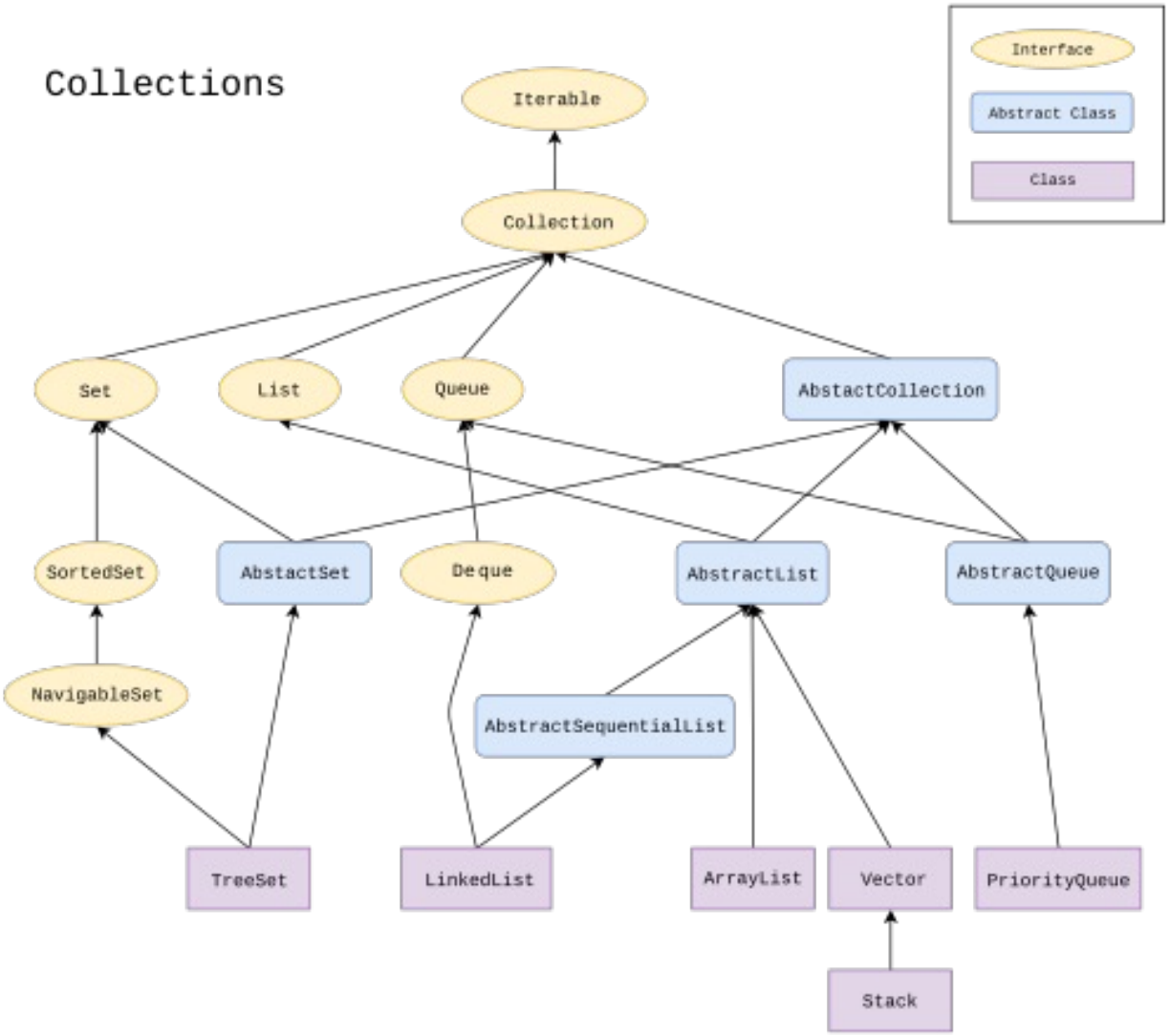
Closed – Effect of poor use



Liskov Substitution Principle

- “’ Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program’” (https://en.wikipedia.org/wiki/SOLID)

Java Collection Framework



Any code that accepts a List should work fine if passed an object of type

- AbstractList
- ArrayList
- Vector
- Stack

Image from https://en.wikipedia.org/wiki/Java_collections_framework, accessed September 18, 2018

Sample code

- **Code uses**

```
private class cell {
        ...
        protected Set<Character> possibleValues = new HashSet<>();
        ...
}
```

Allows us to change our minds on the implementation later without searching _all_ the code for the class name to change.

Rather than

```
private class cell {
        ...
        protected HashSet<Character> possibleValues = new HashSet<>();
        ...
}
```

Locks us in to one implementation.

Design by Contract

- **Preconditions cannot be strengthened by a subtype**
  - **You can't expect more from the subclass than from the superclass**
- **Postconditions cannot be weakened by a subtype**
  - **The outcome of a subclass must be at least as dependable / strong / reliable as the superclass**