Single Responsibility Principle

1. Meaning

Class should have only one responsibility which means class should be highly cohesive and implement strongly related logic. Class implementing feature 1 AND feature 2 AND feature 3 (and so on) violates SRP.

2. Example

```
# BAD

class PlaceOrder
  def initialize(product)
    @product = product
  end

  def run
    # 1. Logic related to verification of
    #    stock availability
    # 2. Logic related to payment process
    # 3. Logic related to shipment process
  end
end
```

```
# GOOD

class PlaceOrder
  def initialize(product)
    @product = product
  end

  def run
    StockAvailability.new(@product).run
    ProductPayment.new(@product).run
    ProductShipment.new(@product).run
  end
end
```

3. How to recognize code smell?

- more than one contextually separated piece of code within single class
- large setup in tests (TDD is very useful when it comes to detecting SRP violation)

4. Benefits

- separated classes responsible for given use case can be now reused in other parts of an application
- separated classes responsible for given use case can be now tested separately

Open/closed Principle

1. Meaning

Class should be open for extension and closed for modification. You should be able to extend class' behavior without the need to modify its implementation (how? Don't modify existing code of class X but write a new piece of code that will be used by class X).

2. Example

```
# BAD

class Logger
  def initialize(logging_form)
    @logging_form = logging_form
  end

  def log(message)
    puts message if @logging_form == "console"
    File.write("logs.txt", message) if @logging_form == "file"
  end
end
```

```
# GOOD

class EventTracker
  def initialize(logger: ConsoleLogger.new)
    @logger = logger
  end

  def log(message)
    @logger.log(message)
  end
end

class ConsoleLogger
  def log(message)
    puts message
  end
end

class FileLogger
  def log(message)
    File.write("logs.txt", message)
  end
end
```

3. How to recognize code smell?

- if you notice class X directly references other class Y from within its code base, it's a sign that class Y should be passed to class X (either through constructor/single method) e.g. through dependency injection
- complex if-else or switch statements

4. Benefits

- class' X functionality can be easily extended with new functionality encapsulated in a separate class without the need to change class' X implementation (it's not aware of introduced changes)
- code is loosely coupled
- injected class Y can be easily mocked in tests

Liskov Substitution Principle

1. Meaning

Extension of open/closed principle stating that new derived classes extending the base class should not change the behavior of the base class (behavior of inherited methods). Provided that if a class Y is a subclass of class X any instance referencing class X should be able to reference class Y as well (derived types must be completely substitutable for their base types.).

2. Example

```
# BAD

class Rectangle
  def initialize(width, height)
    @width, @height = width, height
  end

  def set_width(width)
    @width = width
  end

  def set_height(height)
    @height = height
  end
end

class Square < Rectangle
  # LSP violation: inherited class
  # overrides behavior of parent's methods
  def set_width(width)
    super(width)
    @height = height
  end

  def set_height(height)
    super(height)
    @width = width
  end
end
```

3. How to recognize code smell?

- if it looks like a duck, quacks like a duck but needs batteries for that purpose - it's probably a violation of LSP
- modification of inherited behavior in subclass
- exceptions raised in over-ridden inherited methods

4. Benefits

- avoiding unexpected and incorrect results
- clear distinction between shared inherited interface and extended functionality

Interface Segregation Principle

1. Meaning

Client should not depend on interface/methods which it is not using.

2. Example

```
# BAD

class Car
  def open
  end

  def start_engine
  end

  def change_engine
  end
end

# ISP violation: Driver instance does not make use
# of #change_engine
class Drive
  def take_a_ride(car)
    car.open
    car.start_engine
  end
end

# ISP violation: Mechanic instance does not make use
# of #start_engine
class Mechanic
  def repair(car)
    car.open
    car.change_engine
  end
end
```

3. How to recognize code smell?

- one fat interface implemented by many classes where none of these classes implement 100% of interface's methods. Such fat interface should be split into smaller interfaces suitable for client needs.

4. Benefits

- highly cohesive code
- avoiding coupling between all classes using a single fat interface (once a method in the single fat interface gets updated, all classes - no matter they use this method or not - are forced to update accordingly)
- clear separation of business logic by grouping responsibilities into separate interfaces

Dependency Inversion Principle

1. Meaning

High-level modules (e.g. business logic) should not depend on low-level modules (e.g. database operations or I/O). Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

2. Example

```
# BAD

class EventTracker
  def initialize
    # An instance of low-level class ConsoleLogger
    # is directly created inside high-level
    # EventTracker class which increases class'
    # coupling
    @logger = ConsoleLogger.new
  end

  def log(message)
    @logger.log(message)
  end
end
```

```
# GOOD

class EventTracker
  def initialize(logger: ConsoleLogger.new)
    # Use dependency injection as in closed/open
    # principle.
    @logger = logger
  end

  def log(message)
    @logger.log(message)
  end
end
```

3. How to recognize code smell?

- instantiation of low-level modules in high-level ones
- calls to class methods of low-level modules/classes

4. Benefits

- increase reusability of higher-level modules by making them independent of lower-level modules
- injected class can be easily mocked in tests