Design Patterns

Iterator

Code Smells

COMP3607
Object Oriented Programming II

Week 8

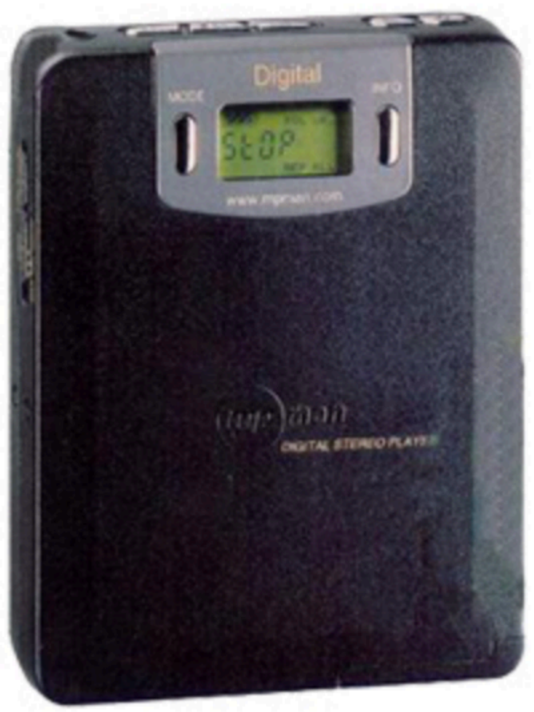
Outline

- Design Patterns
  - Iterator
- Code Smells
  - Change Preventers
  - Dispensibles

Mp3 Players

1998

First Launch



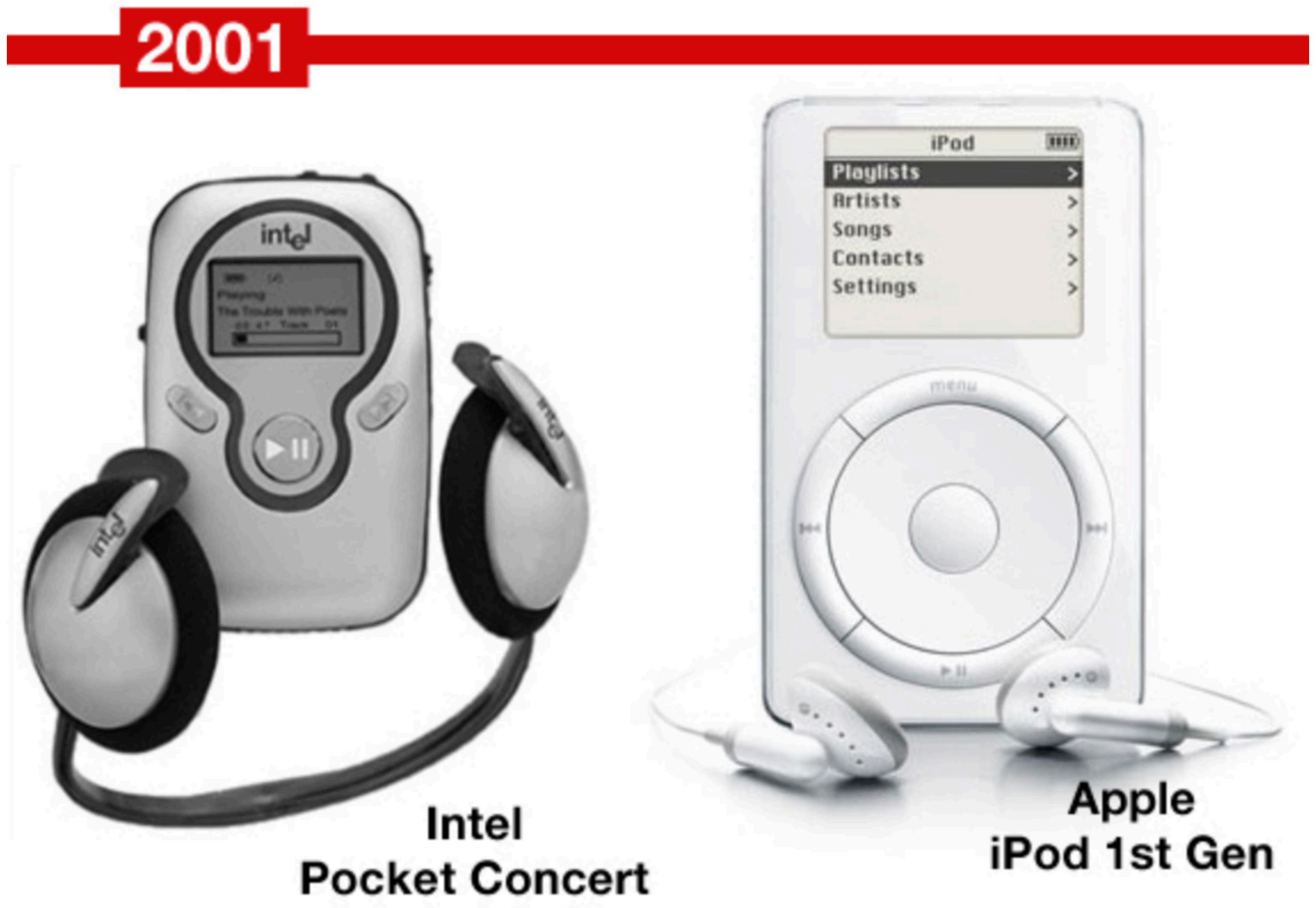
Elger Labs MPMAN F10

Diamond RIO PMP300

Mp3 Players

Menu Options

Wheel Navigation



Mp3 Players

2004

Colour screens

Creative Zen Media Center

iRiver H300

https://www.pcworld.com/article/174725/evolution_of_the_mp3_player.html

Mp3 Players

2007

Touch screens

Apple
iPod touch 1st Gen

Samsung
P2

Mp3 Players

https://www.pcworld.com/article/174725/evolution_of_the_mp3_player.html

Iterator in the Real World

MP3 player control is a good example of an iterator.

The user doesn't mind how to view their list of songs, once they get to see them somehow.

In older mp3 players, this was done using simple forward and back buttons.

With the iPod this changed to the wheel navigation concept.

The iPhone moves this on further to use swipe movements.

Nevertheless, the same idea is provided by all interfaces - a way to iterate through your music collection.

https://dzone.com/articles/design-patterns-iterator

Iterator Design Pattern

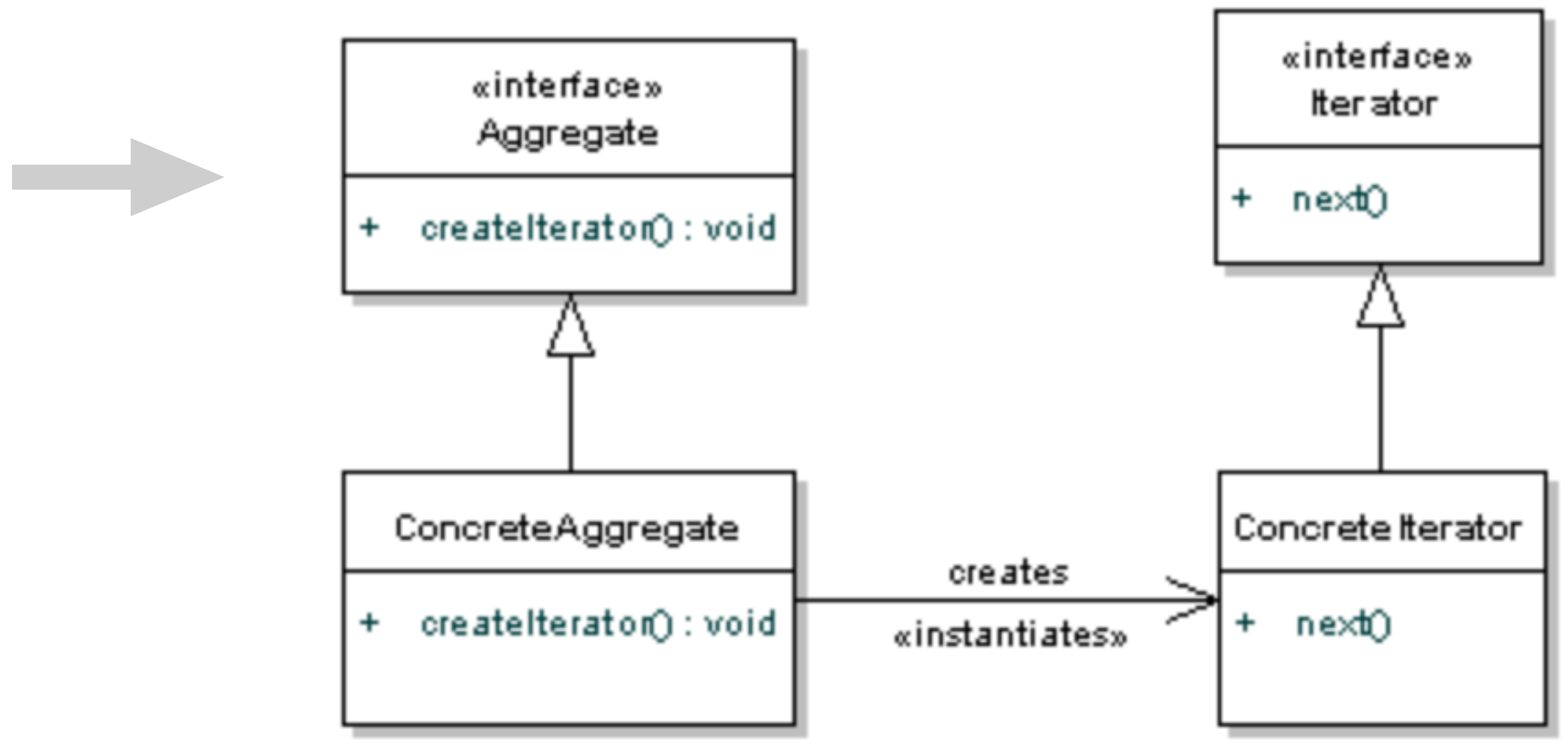
The Iterator pattern is known as a behavioural pattern, as it's used to manage algorithms, relationships and responsibilities between objects.

The definition of Iterator as provided in the original Gang of Four book on Design Patterns states:

- Provides a way to access the elements of an aggregate object without exposing its underlying representation.

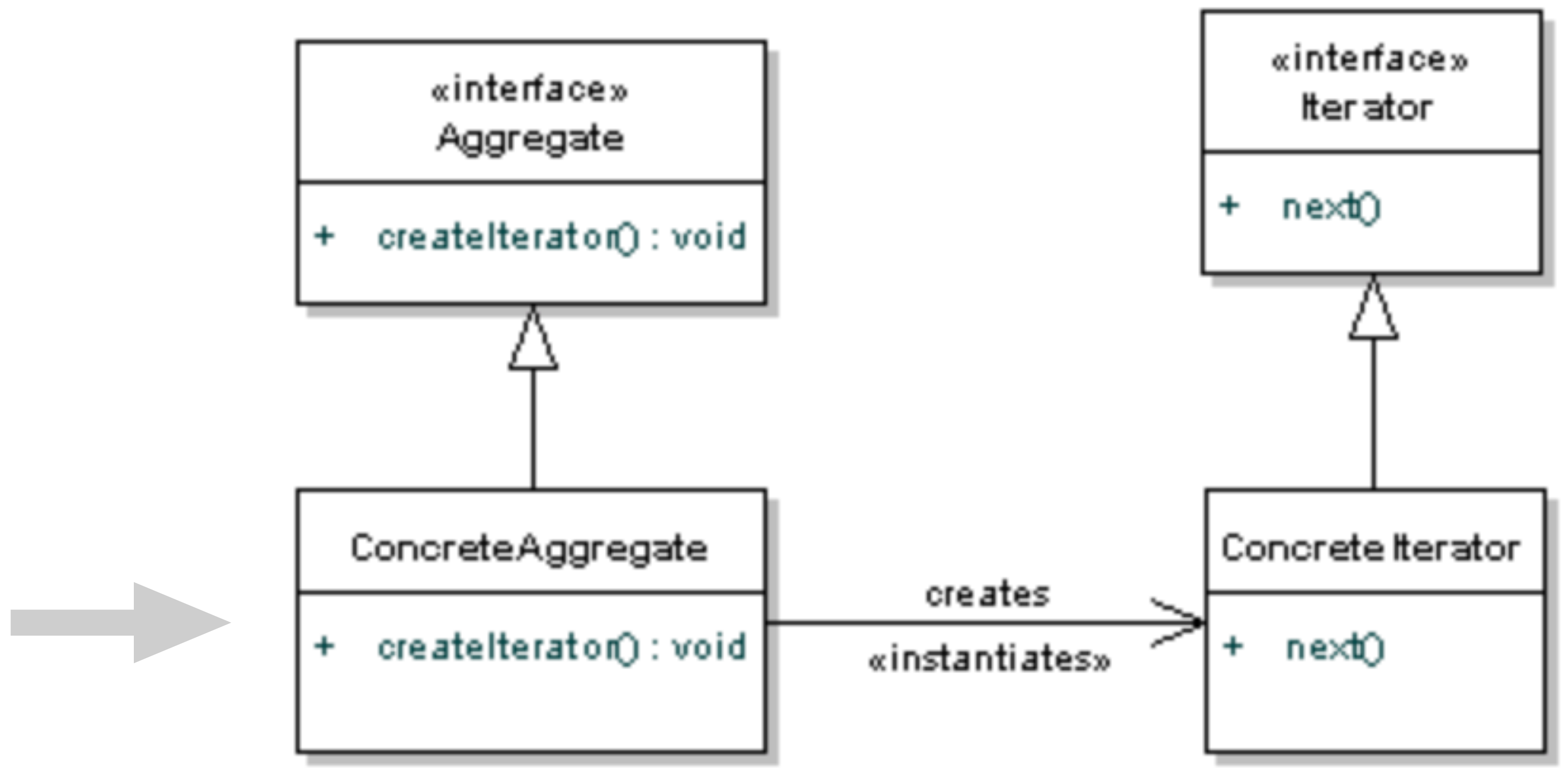
https://dzone.com/articles/design-patterns-iterator

Iterator Pattern Structure



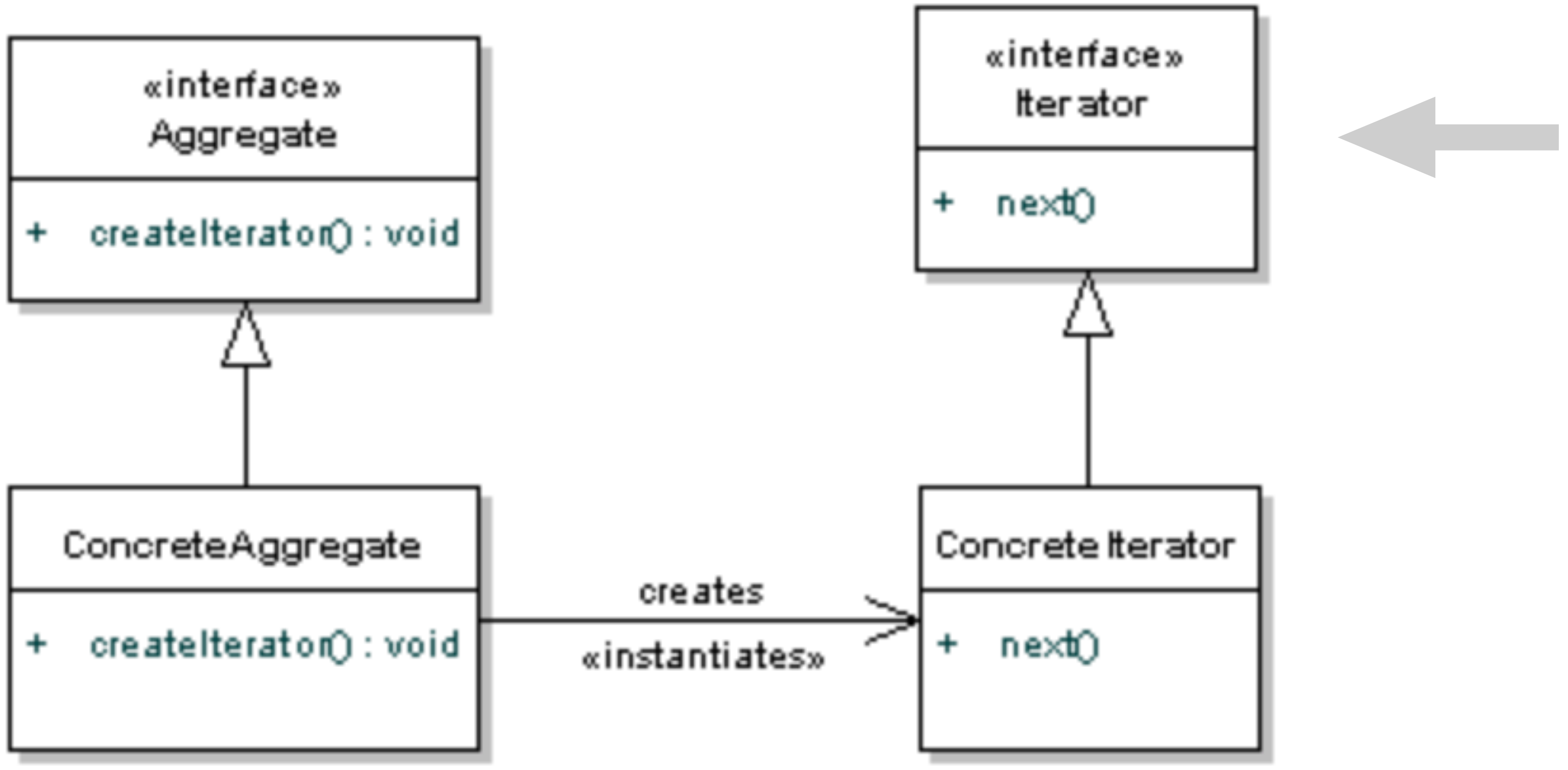
The **Aggregate** defines an interface for the creation of the Iterator object.

Iterator Pattern Structure



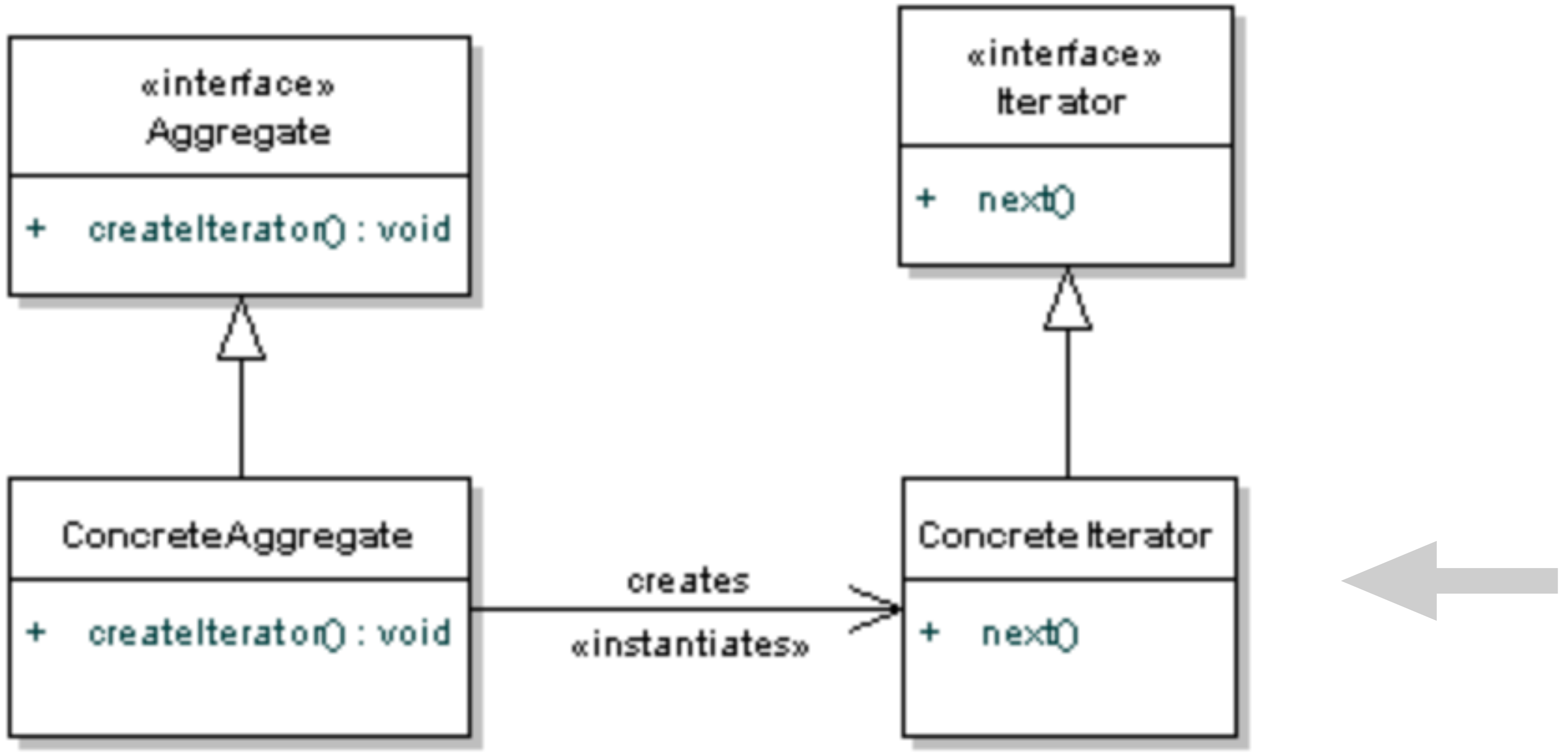
The ConcreteAggregate implements this interface, and returns an instance of the ConcreteIterator.

Iterator Pattern Structure



The **Iterator** defines the interface for access and traversal of the elements.

Iterator Pattern Structure



The **ConcreteIterator** implements this interface while keeping track of the current position in the traversal of the Aggregate.

Problem Scenario

This example uses a collection of books and an iterator to iterate through the collection. The main actors are:

- IIterator - This interface represent the AbstractIterator, defining the iterator
- BookIterator - This is the implementation of Iterator(implements the IIterator interface)
- IContainer - This is an interface defining the Agregate
- BooksCollection - An implementation of the collection

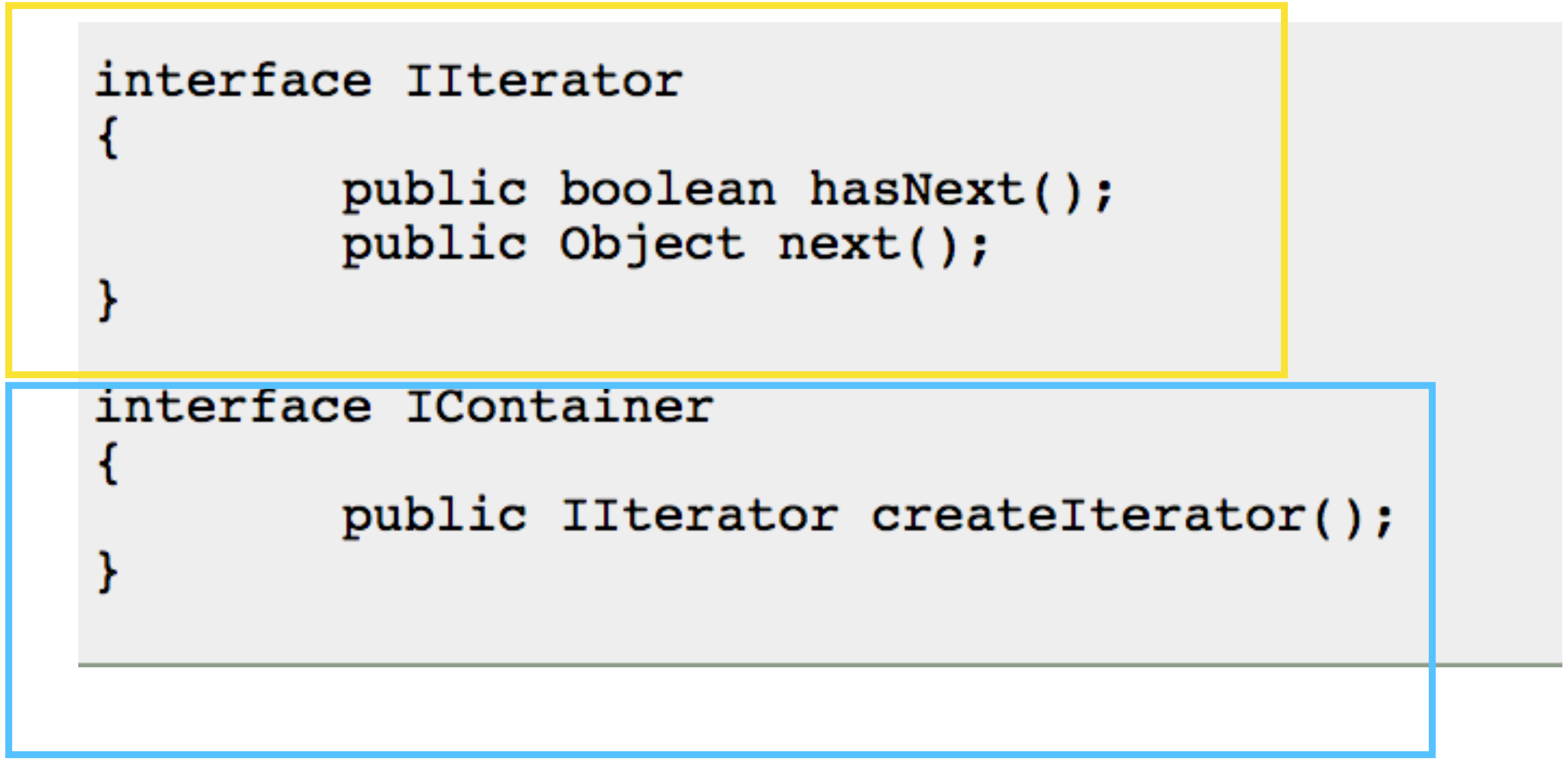
https://www.oodesign.com/iterator-pattern.html

8

Code Example

Iterator

Aggregate



```
interface IIterator
{
        public boolean hasNext();
        public Object next();
}
interface IContainer
{
        public IIterator createIterator();
}
```

https://www.oodesign.com/iterator-pattern.html

Code Example

Concrete Aggregate

```
class BooksCollection implements IContainer
{
        private String m_titles[] = {"Design Patterns","1","2","3","4"};

        public IIterator createIterator()
        {
                BookIterator result = new BookIterator();
                return result;
        }
```

Concrete Iterator

```
        private class BookIterator implements IIterator
        {
                private int m_position;

                public boolean hasNext()
                {
                        if (m_position < m_titles.length)
                                return true;
                        else
                                return false;
                }
                public Object next()
                {
                        if (this.hasNext())
                                return m_titles[m_position++];
                        else
                                return null;
                }
        }
}
```

https://www.oodesign.com/iterator-pattern.html

Applicability: Iterator Pattern

This pattern is useful when you need access to elements in a set without access to the entire representation.

When you need a uniform traversal interface, and multiple traversals may happen across elements, iterator is a good choice.

It also makes your code much more reasonable, getting rid of the typical for loop syntax across sections of your codebase.

Code Smells

“A surface indication that usually corresponds to a deeper problem in the software system” - Martin Fowler

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

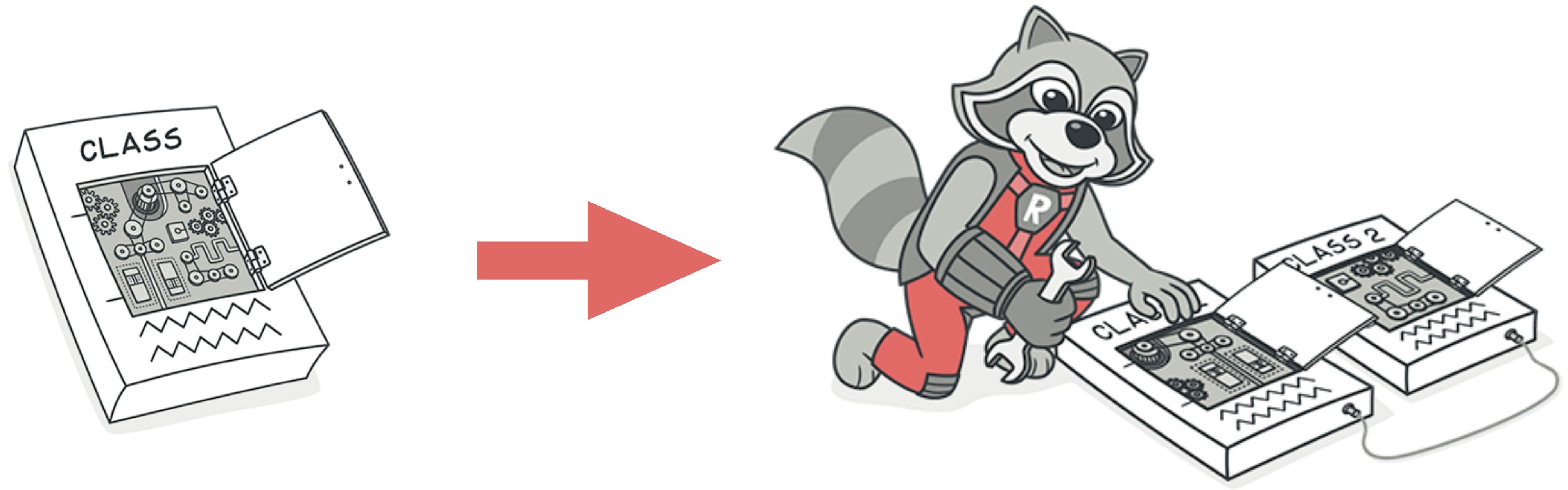
Program development becomes much more complicated and expensive as a result.

For example:

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Divergent Change

It is when a class is commonly changed in different ways for different reasons and suffers many kinds of changes. So, ideally, you should have a one-to-one link between common changes and classes.

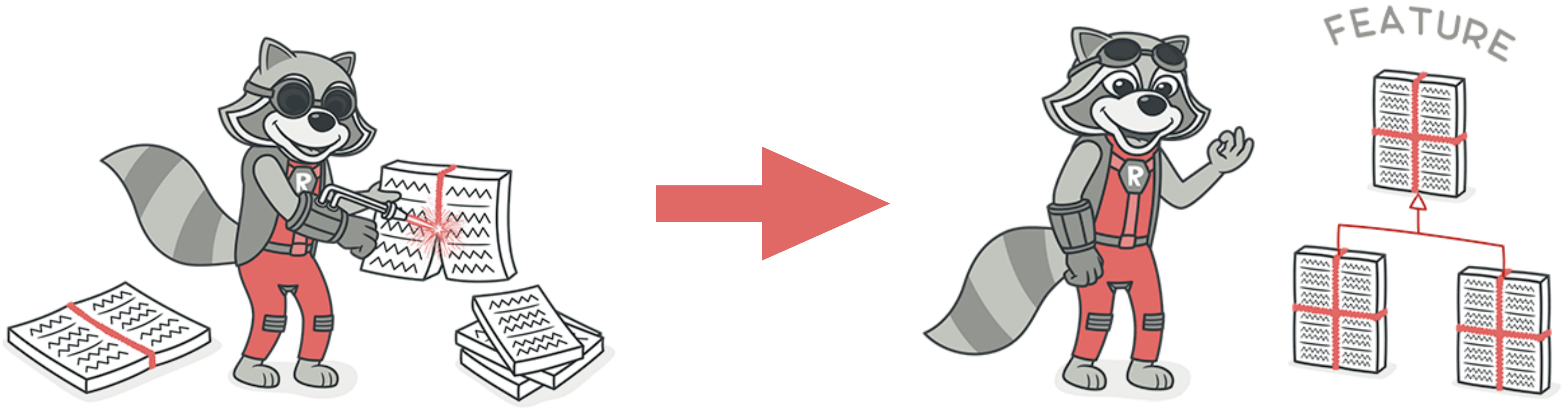


https://refactoring.guru/smells/divergent-change

Shotgun Surgery

It is basically when you want to make a kind of change, you need to make a lot of little changes to a lot of different classes.

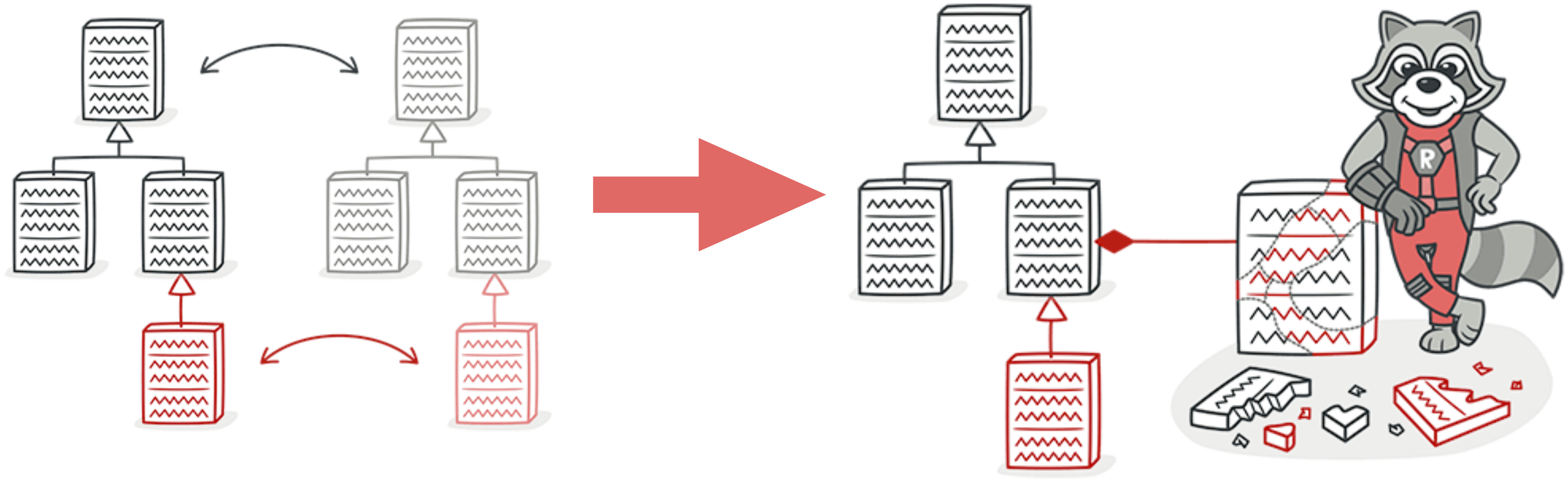
The problem is that when the changes are all over the place, they are hard to find, and it’s easy to miss an important change.



https://refactoring.guru/smells/shotgun-surgery

Parallel Inheritance Hierarchies

Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class



https://refactoring.guru/smells/parallel-inheritance-hierarchies

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

For example:

- Comments
- Duplicate Code
- Lazy Class
- Data Class
- Dead Code
- Speculative Generality.

https://refactoring.guru/refactoring/smells/dispensables

Comments

Where comments are re-iterating what can be read by a developer, they may provide little value, especially when they have not been updated and no longer reflect the intent of the current code.

Rather than adding a comment to clarify a piece of code, think about whether the code can be refactored such that the comment is no longer needed.

It may be possible to provide a more descriptive name that provides the same clarity as the comment, meaning the comment can disappear, resulting in more intuitive and readable code.

Duplicate Code

When developer fixes a bug, but same symptoms are faced again later on, this can be the result of code duplication, and a bug being fixed in one occurrence of the imperfect code but not in the duplicated versions.

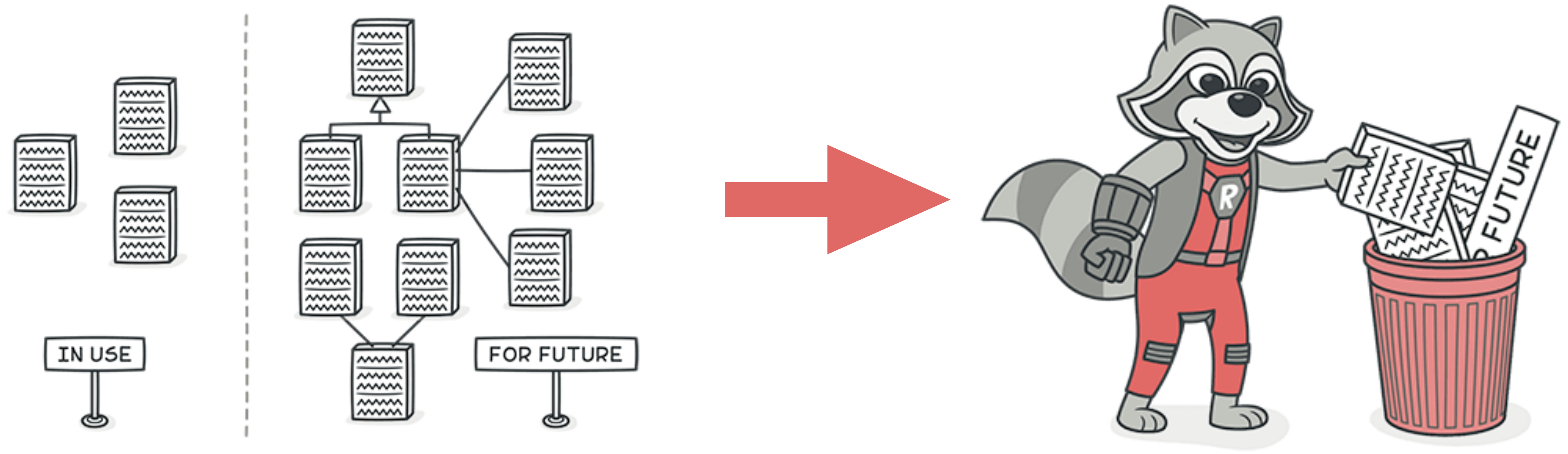
This poses an overhead in terms of maintenance.

When developers are not aware of the duplication, they only know to fix the occurrence they have come across.

Take care of the repeated code blocks and extract them out into a single place – don’t repeat yourself!

Speculative Generality

Sometimes code is created "just in case" to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.



https://refactoring.guru/smells/speculative-generality

References

- Design Patterns: online reading resources
  - https://dzone.com/articles/design-patterns-iterator
  - https://sourcemaking.com/design_patterns/iterator
- Code Smells
  - https://refactoring.guru/refactoring/smells/change-preventers