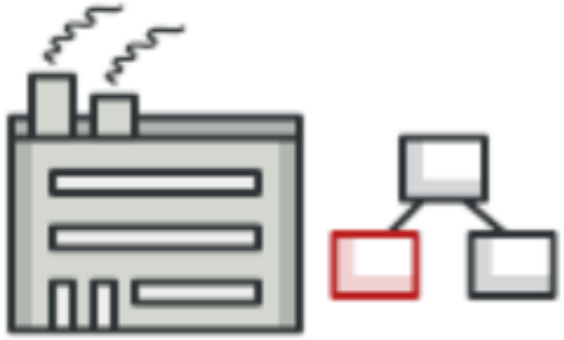
Creational Design Patterns

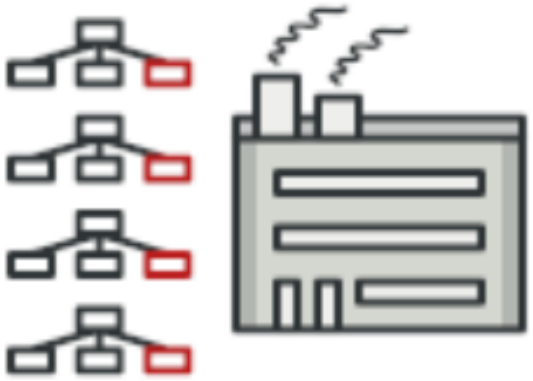


Factory Method

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Credit:
refactoring.guru/

Creational Design Patterns



Abstract Factory

Lets you produce families of related objects without specifying their concrete classes.

Credit:
refactoring.guru/

Creational Design Patterns

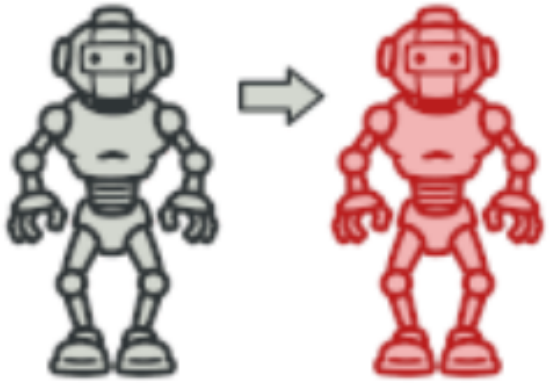
Builder

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Credit:
refactoring.guru/

REFACTORING
·GURU·

Creational Design Patterns

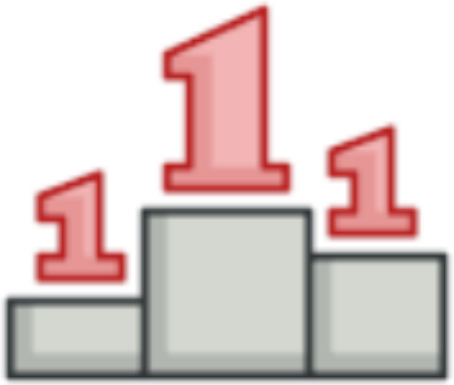


Prototype

Lets you copy existing objects without making your code dependent on their classes.

Credit:
refactoring.guru/

Creational Design Patterns



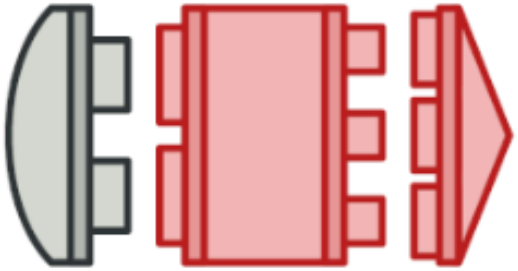
Singleton

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Credit:
refactoring.guru/

REFACTORING
·GURU·

Structural Design Patterns



Adapter

Allows objects with incompatible interfaces to collaborate.

Credit:
refactoring.guru/

Structural Design Patterns

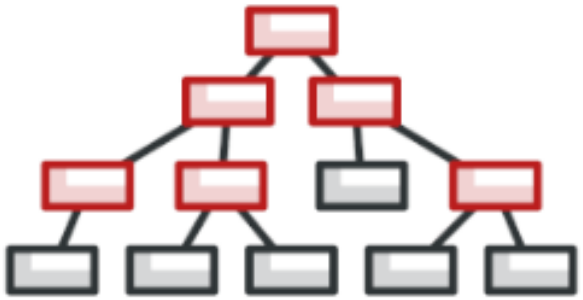
Bridge

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Credit:
refactoring.guru/

REFACTORING
·GURU·

Structural Design Patterns

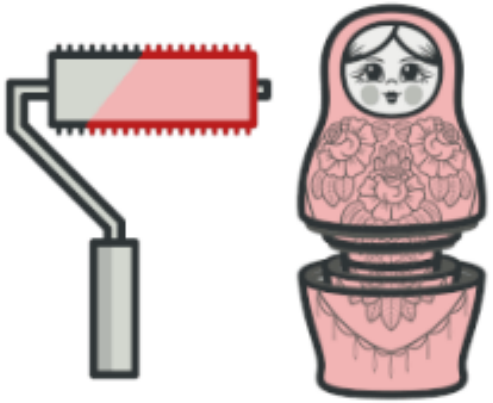


Composite

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Credit:
refactoring.guru/

Structural Design Patterns



Decorator

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Credit:
refactoring.guru/

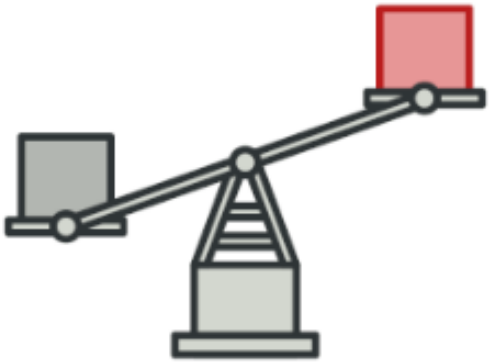
Structural Design Patterns

Facade

Provides a simplified interface to a library, a framework, or any other complex set of classes.

Credit:
refactoring.guru/

Structural Design Patterns

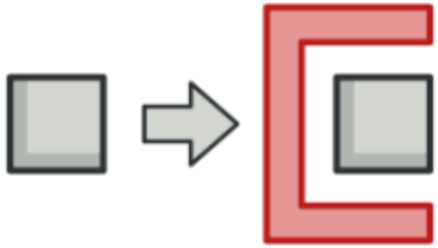


Flyweight

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Credit:
refactoring.guru/

Structural Design Patterns



Proxy

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Credit:
refactoring.guru/

Behavioral Design Patterns

Chain of Responsibility

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Credit:
refactoring.guru/

Behavioral Design Patterns

Command

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you pass requests as a method arguments, delay or queue a request's execution, and support undoable operations.

Credit:
refactoring.guru/

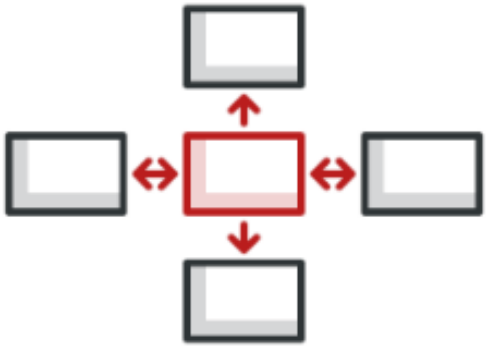
Behavioral Design Patterns

Iterator

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Credit:
refactoring.guru/

Behavioral Design Patterns

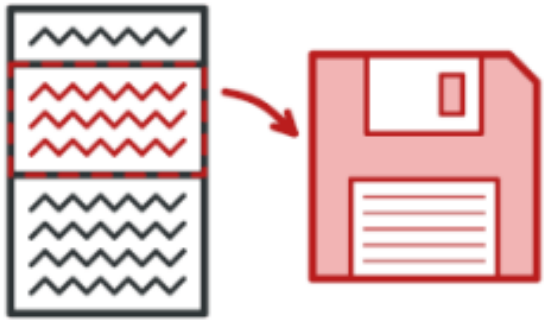


Mediator

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Credit:
refactoring.guru/

Behavioral Design Patterns

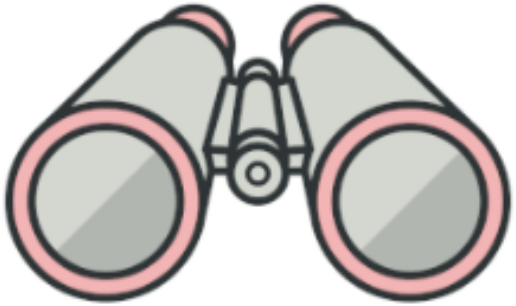


Memento

Lets you save and restore the previous state of an object without revealing the details of its implementation.

Credit:
refactoring.guru/

Behavioral Design Patterns

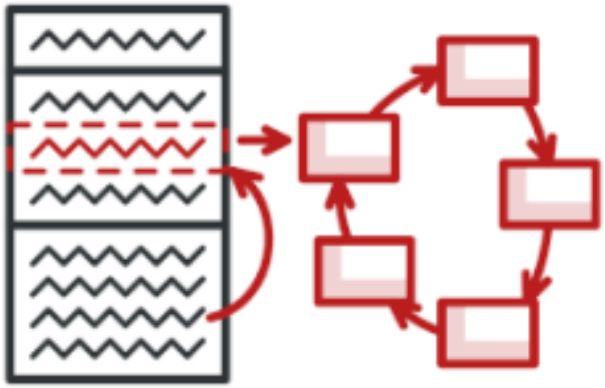


Observer

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Credit:
refactoring.guru/

Behavioral Design Patterns

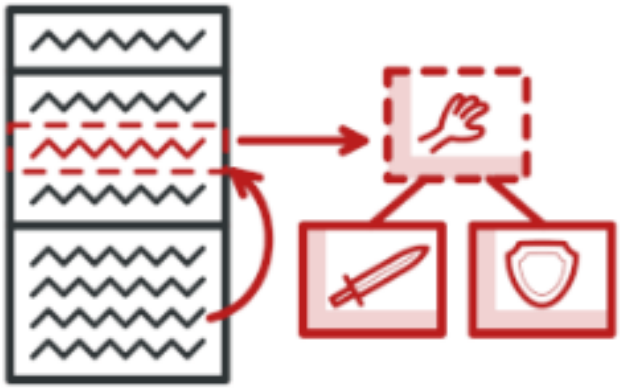


State

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Credit:
refactoring.guru/

Behavioral Design Patterns



Strategy

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Credit:
refactoring.guru/

Behavioral Design Patterns

Template Method

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Credit:
refactoring.guru/

Behavioral Design Patterns

Visitor

Lets you separate algorithms from the objects on which they operate.

Credit:
refactoring.guru/

Composing Methods

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand—and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

- Extract Method
- Inline Method
- Extract Variable
- Inline Temp
- Replace Temp with Query
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Substitute Algorithm

Credit:
refactoring.guru/

Moving Features between Objects

Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.

These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- § Move Method
- § Move Field
- § Extract Class
- § Inline Class
- § Hide Delegate
- § Remove Middle Man
- § Introduce Foreign Method
- § Introduce Local Extension

Credit:
refactoring.guru/

Organizing Data

These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.

- Change Value to Reference
- Change Reference to Value
- Duplicate Observed Data
- Self Encapsulate Field
- Replace Data Value with Object
- Replace Array with Object
- Change Unidirectional Association to Bidirectional
- Change Bidirectional Association to Unidirectional
- Encapsulate Field
- Encapsulate Collection
- Replace Magic Number with Symbolic Constant
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Replace Type Code with State/Strategy
- Replace Subclass with Fields

Credit:
refactoring.guru/

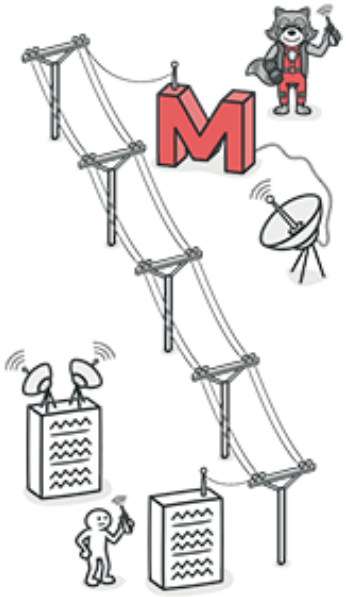
REFACTORING
·GURU·

Simplifying Conditional Expressions

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

- Consolidate Conditional Expression
- Consolidate Duplicate Conditional Fragments
- Decompose Conditional
- Replace Conditional with Polymorphism
- Remove Control Flag
- Replace Nested Conditional with Guard Clauses
- Introduce Null Object
- Introduce Assertion

Credit:
refactoring.guru/

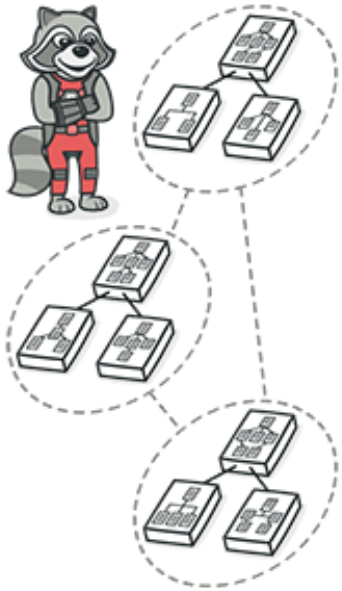


Simplifying Method Calls

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

- § Add Parameter
- § Remove Parameter
- § Rename Method
- § Separate Query from Modifier
- § Parameterize Method
- § Introduce Parameter Object
- § Preserve Whole Object
- § Remove Setting Method
- § Replace Parameter with Explicit Methods
- § Replace Parameter with Method Call
- § Hide Method
- § Replace Constructor with Factory Method
- § Replace Error Code with Exception
- § Replace Exception with Test

Credit:
refactoring.guru/



Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

- § Pull Up Field
- § Pull Up Method
- § Pull Up Constructor Body
- § Push Down Field
- § Push Down Method
- § Extract Subclass
- § Extract Superclass
- § Extract Interface
- § Collapse Hierarchy
- § Form Template Method
- § Replace Inheritance with Delegation
- § Replace Delegation with Inheritance

Credit:
refactoring.guru/

REFACTORING
·GURU·