Design Patterns Workshop
with
Tushar Joshi

Design Patterns Workshop

Tushar Joshi

Table of Contents

Licence

[1] Pattern from Martin Fowlers Catalog of Patterns of Entreprise Application Architecture

Preface

What is this manual about?

This manual is compilation of definition, discussion and examples of 23 Design Patterns mentioned in the GoF Design Patterns book from 1994 by Kent Beck, Eric Gamma, et all.

The compilation of this manual started as a supplementary documentation for Design Patterns Workshop. This is an effort to compile enough available information relevant to the topic so it can be referred as a handy reference.

The manual has been updated from the suggestions and discussions which happened in the Design Patterns Workshop conducted at Persistent Systems.

The examples referred in this manual are available at GitHub repository for the Design Patterns Workshop also at Gitlab and Bitbucket

Major Goal of this manual

To maintain minimum one example for each design pattern from the initial 23 design patterns. The examples are designed to be as near to the production code which can be used in projects. Usage of apple, banana like examples which are created just to explain the design patterns are avoided.

Basic examples and many examples are already available all over the internet and the books and articles mentioned in the reference section.

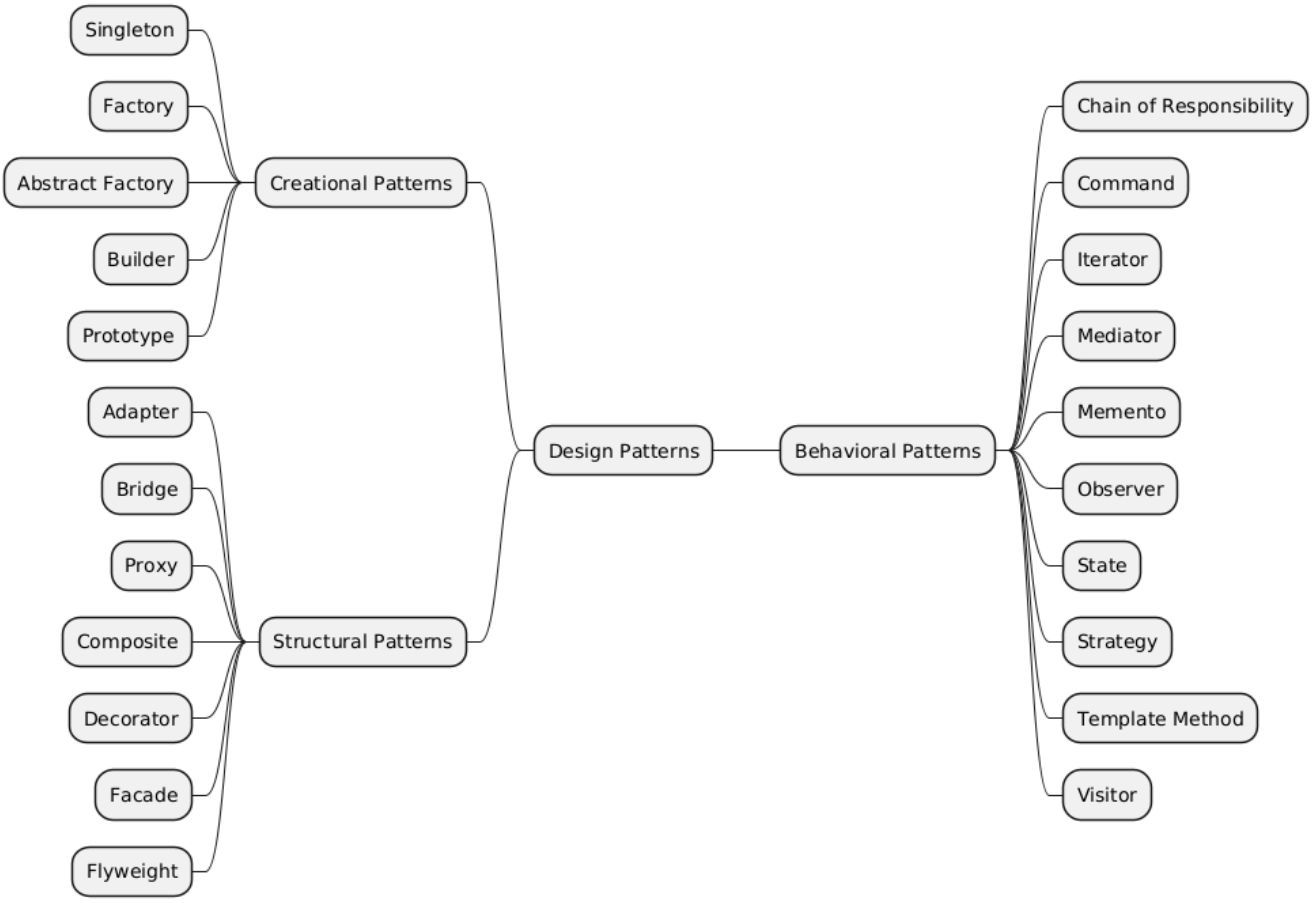
Dedication

To my mentor Rahul Ekbote, his vision and support made this work possible.

Introduction

Patterns

Patterns is a concept introduced by Christopher Alexander,[1 - Licence] and is widely accepted in the software community to document design constructs which are used to build software systems. Patterns provide a structured way of looking at a problem space with the solutions which are seen multiple times and proven.

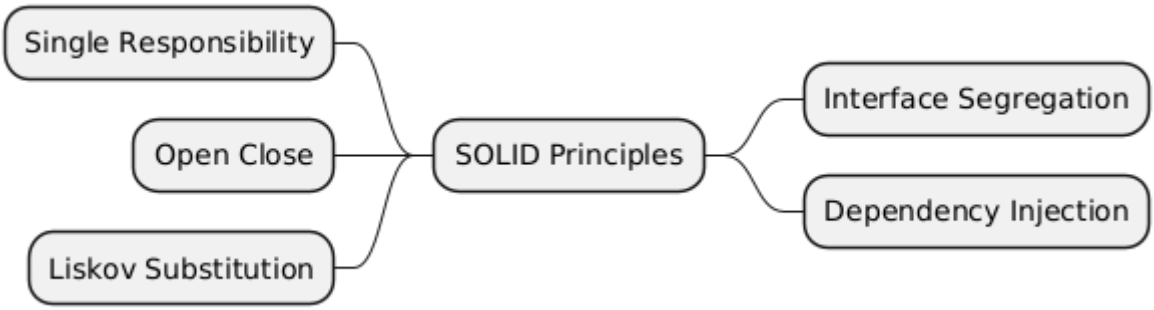


Let's get started.

[0] Christopher Wolfgang Alexander is a British-American architect and design theorist. His theories about the nature of human-centered design have affected fields beyond architecture, including urban design, software, sociology and others. Alexander is regarded as the father of the pattern language movement.

SOLID Principles

SOLID Principles Introduction



Single Responsibility Principle

The single responsibility principle instructs developers to write code that has one and only one reason to change. If a class has more than one reason to change, it has more than one responsibility. Classes with more than a single responsibility should be broken down into smaller classes, each of which should have only one responsibility and reason to change.

All In One

This example violates the single responsibility principle by doing everything at one place. Such programs are usually written as POC and temporary code, but the Ball Of Mud pattern prevails many times and program written for temporary purposes ends in real world just because it works and provides certain functionality.

```
public class Main {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        in.close();

        String greeting = "Hello " + name + " Greetings and Welcome!";

        System.out.println(greeting);
    }
}
```

Modular Design with Single Responsibility

Separating the responsibilities into multiple components not only abides by single responsibility principle, it also makes the code unit smaller, easy to read and maintainable.

```
public class Greeter {
    private NameProvider nameProvider;
    private MessageDisplayer displayer;

    public Greeter(NameProvider nameProvider, MessageDisplayer displayer) {
        this.nameProvider = nameProvider;
        this.displayer = displayer;
    }

    public void greet() {
        String greeting = "Hello "
            + nameProvider.getName()
            + " Greetings and Welcome!";
        displayer.display(greeting);
    }
}
```

```
public class NameProvider {
    public String getName() {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter your name: ");
        String name = in.nextLine();
        in.close();
        return name;
    }
}
```

```
public class MessageDisplayer {
    public void display(String message) {
        System.out.print(message);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        NameProvider nameProvider = new NameProvider();
        MessageDisplayer displayer = new MessageDisplayer();
        Greeter greeter = new Greeter(nameProvider, displayer);
        greeter.greet();
    }
}
```

```
class GreeterTest {

    @Test
```

```
    void test_greet() {
        MockNameProvider nameProvider = new MockNameProvider();
        MockDisplayer displayer = new MockDisplayer();
        Greeter greeter = new Greeter(nameProvider, displayer);
        greeter.greet();

        assertEquals(
            "Hello MockName Greetings and Welcome!",
            displayer.getMessage()
        );
    }

    private class MockNameProvider extends NameProvider {
        public String getName() {
            return "MockName";
        }
    }

    private class MockDisplayer extends MessageDisplayer {
        private String message;

        public String getMessage() {
            return message;
        }

        public void display(String message) {
            this.message = message;
        }
    }
}
```

Open Close Principle

Bertrand Mayer definition

From his 1988 book, Object-Oriented Software Construction (Prentice Hall), defined the open/closed principle (OCP) as follows:

Software entities should be open for extension, but closed for modification.

Robert C. Martin definition

“Open for extension.” This means that the behavior of the module can be extended. As the requirements of the application change, we are able to extend the module with new behaviors that satisfy those changes. In other words, we are able to change what the module does.

“Closed for modification.” Extending the behavior of a module does not result in changes to the source or binary code of the module. The binary executable version of the module, whether in a linkable library, a DLL, or a Java .jar, remains untouched.

—Robert C. Martin, Agile Software Development: Principles, Patterns, and Practices (Prentice Hall, 2003)

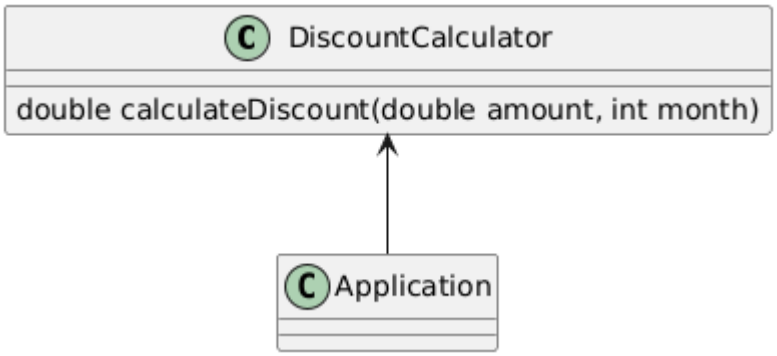
Open Close Principle advocates class to be open for extension but closed for modifications. To enhance the functionality the design should allow additions of new functionality by adding new files to the code, instead of modifying an existing file.

Switch Unclean Example

The logic in the example below in `DiscountCalculator` is based on month number. For different months we need discount to be calculated with some rules. For addition of new rules or modification of any existing rule we will have to change the file `DiscountCalculator` and hence this file violates the open close principle.

Just to change one rule if we touch the file `DiscountCalculator` we will have to regression test all the rules to make sure nothing is broken in other parts which are also in the same file.

UML Representation




```
public class DiscountCalculator {

    public double calculateDiscount(double amount, int month) {
        double discount = 0;
        switch(month) {
            case 1:
            case 2:
                discount = amount * 0.1;
                break;
            case 8:
                discount = amount * 0.2;
                break;
            default:
                discount = 0;
                break;
        }
        return discount;
    }
}
```

```
class DiscountCalculatorTest {

    private DiscountCalculator target;

    @BeforeEach
    public void beforeEach() {
        target = new DiscountCalculator();
    }

    @Test
    void verify_janfeb_discount_10() {
        double discount = target.calculateDiscount(1000, 1);
        assertEquals(100, discount);
    }

    @Test
    void verify_aug_discount_10() {
        double discount = target.calculateDiscount(1000, 8);
        assertEquals(200, discount);
    }

    @Test
    void verify_other_discount_10() {
        double discount = target.calculateDiscount(1000, 6);
        assertEquals(0, discount);
    }
}
```

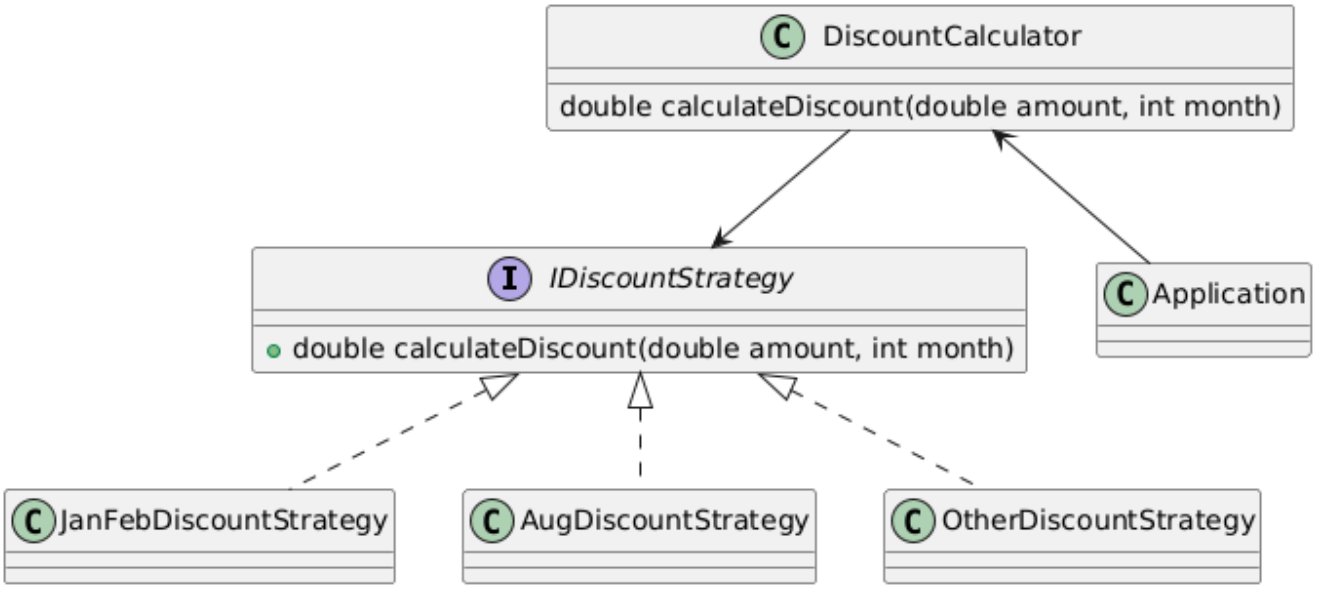
Switch clean version

In this version we extracted the logic of calculating the discount into separate classes implementing same interface. The map of all such strategies will maintain what rule we want to use for certain month.

If we need to change an existing rule only one existing class needs changes and the rest of the rules need no testing as they were not changed.

For addition of new rule we need to add a new class implementation of strategy and add it to the map. Again retesting of existing rules is not needed, as `DiscountCalculator` class is now open to extension but closed to modification.

UML Representation




```
public class DiscountCalculator {
    public double calculateDiscount(double amount, int month) {
        IDiscountStrategy strategy = DiscountStrategyMap.get(month); ①
        return strategy.calculateDiscount(amount, month);
    }
}
```

① Get strategy object for given month

```
public interface IDiscountStrategy {
    double calculateDiscount(double amount, int month);
}
```

```
public class DiscountStrategyMap {
    private static Map<Integer, IDiscountStrategy> strategyMap = createStrategies();

    private static Map<Integer, IDiscountStrategy> createStrategies() {
        Map<Integer, IDiscountStrategy> map = new HashMap<>();
        IDiscountStrategy janFebStrategy = new JanFebDiscountStrategy();
        IDiscountStrategy augStrategy = new AugDiscountStrategy();
        IDiscountStrategy otherStrategy = new OtherDiscountStrategy();
        map.put(1, janFebStrategy);
        map.put(2, janFebStrategy);
        map.put(3, otherStrategy);
        map.put(4, otherStrategy);
        map.put(5, otherStrategy);
        map.put(6, otherStrategy);
        map.put(7, otherStrategy);
        map.put(8, augStrategy);
        map.put(9, otherStrategy);
        map.put(10, otherStrategy);
        map.put(11, otherStrategy);
```

```
        map.put(12, otherStrategy);

        return map;
    }

    public static IDiscountStrategy get(int month) {
        return strategyMap.get(month);
    }
}
```

```
public class JanFebDiscountStrategy implements IDiscountStrategy {
    public double calculateDiscount(double amount, int month) {
        return amount * 0.1;
    }
}
```

```
public class AugDiscountStrategy implements IDiscountStrategy {
    public double calculateDiscount(double amount, int month) {
        return amount * 0.2
        ;
    }
}
```

```
public class OtherDiscountStrategy implements IDiscountStrategy {
    public double calculateDiscount(double amount, int month) {
        return 0.0;
    }
}
```

Liskov Substitution Principle

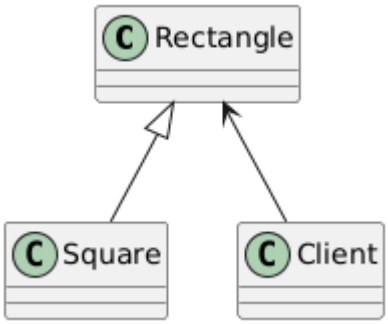
The component/object must honor the public contract it promises

Barbara Liskov Definition

The Liskov Substitution principle was introduced by Barbara Liskov in her conference keynote “Data abstraction” in 1987.

If S is a subtype of T, then objects of type T may be replaced with objects of type S, without breaking the program.

UML Representation




```
public class Rectangle {
    private int length;
    private int breadth;

    public int getLength() {
        return length;
    }

    public void setLength(int length) {
        this.length = length;
    }

    public int getBreadth() {
        return breadth;
    }

    public void setBreadth(int breadth) {
        this.breadth = breadth;
    }

    public int getArea() {
        return length * breadth;
    }
}
```

```
public class Square extends Rectangle {

    public void setLength(int length) {
        super.setLength(length);
        super.setBreadth(length);
    }

    public void setBreadth(int breadth) {
        super.setLength(breadth);
        super.setBreadth(breadth);
    }
}
```

```
public class AreaCalculator {

    public static int calculateArea(Rectangle rectangle) {
        rectangle.setLength(2);
        rectangle.setBreadth(3);

        return rectangle.getArea();
    }
}
```

```
class AreaCalculatorTest {

    @Test
    void test_calculate_area() {
        Rectangle rectangle = new Square();
        int area = AreaCalculator.calculateArea(rectangle); // 2 * 3 = 6

        assertEquals(9, area); // violation of LSP
    }
}
```

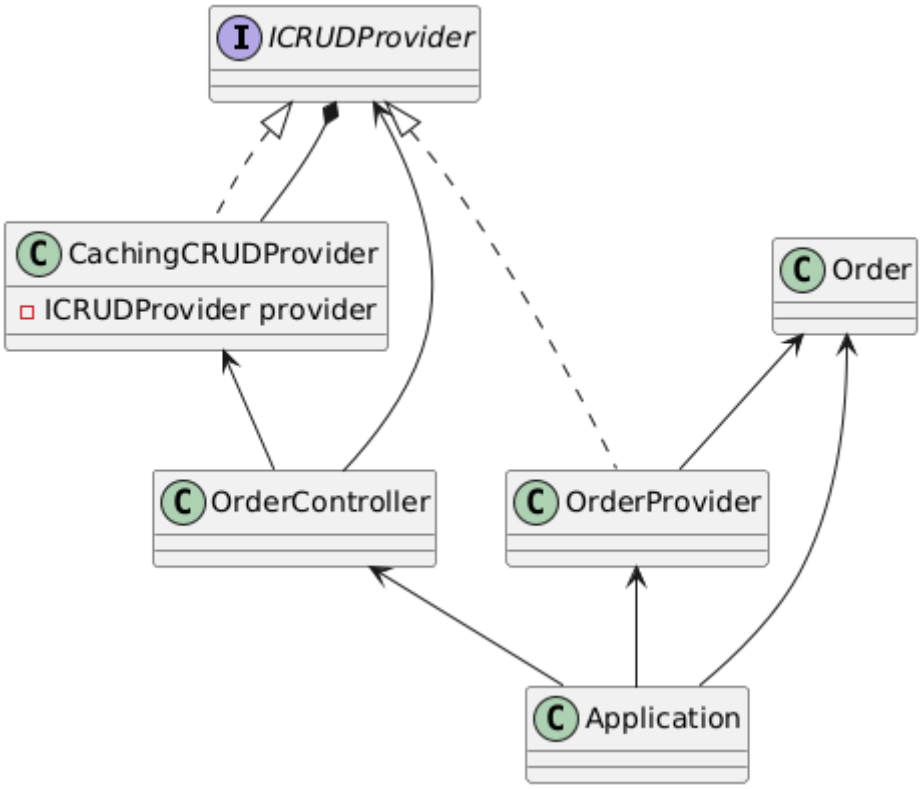
Interface Segregation Principle

Clients should not be forced to depend upon interfaces that they do not use.

Example without ISP

The following example contains an interface having four methods. Any implementation of the `ICRUDProvider` interface need to implement all four methods. Even if only one method is needed for the decoration `CachingCRUDProvider` needs to write delegating methods. This violated the interface segregation principle. The class has boilerplate code and unnecessary test cases need to be written for this class.

UML representation




```
public interface ICRUDProvider<T> {
    void create(T item);
    T read(int id);
    void update(T item);
    void delete(T item);
}
```

```
public class CachingCRUDProvider<T> implements ICRUDProvider<T> {
    private ICRUDProvider<T> provider;
    private Map<Integer, T> itemMap = new HashMap<>();

    public CachingCRUDProvider(ICRUDProvider<T> provider) {
        this.provider = provider;
    }

    public void create(T item) {
        provider.create(item);
    }

    public T read(int id) {
        T item = itemMap.get(id);
        if( item == null ) {
            item = provider.read(id);
            itemMap.put(id, item);
        }
        return item;
    }
```

```
    public void update(T item) {
        provider.update(item);
    }

    public void delete(T item) {
        provider.delete(item);
    }
}
```

```
public class Order {
    private int id;
    private double amount;

    public Order(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }
}
```

```
public class OrderProvider<Order> implements ICRUDProvider<Order> {
    public void create(Order item) {
        // create logic
    }

    public Order read(int id) {
        return (Order)(new com.company.example.Order(id));
    }

    public void update(Order item) {
        // update logic
    }

    public void delete(Order item) {
        // delete logic
    }
}
```

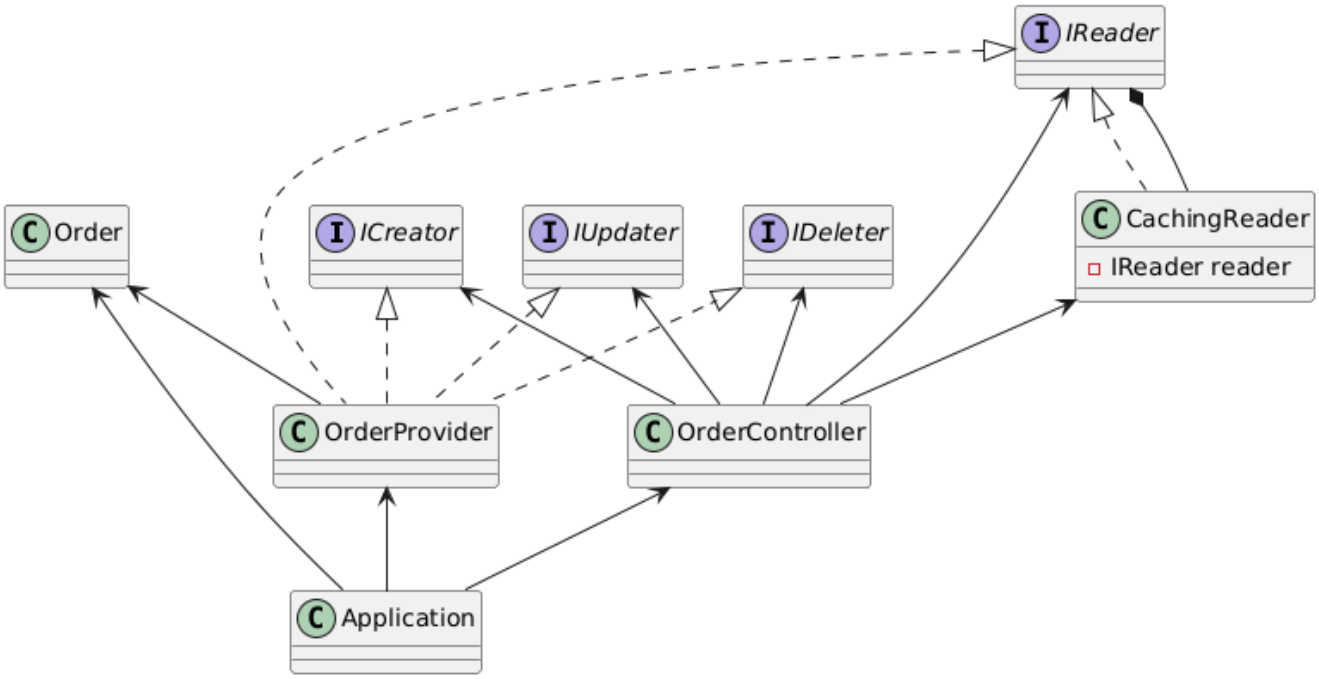
```
public class OrderController {
    private ICRUDProvider<Order> provider;

    public OrderController(ICRUDProvider<Order> provider) {
        this.provider = new CachingCRUDProvider(provider);
    }

    public Order getOrder(int id) {
        return provider.read(id);
    }
}
```

```
class OrderControllerTest {

    @Test
    void test_getOrder() {
        OrderProvider provider = new OrderProvider();
        OrderController orderController = new OrderController(provider);

        Order order1 = orderController.getOrder(1);
        assertEquals(1, order1.getId());

        Order order2 = orderController.getOrder(1);
        assertEquals(1, order1.getId());

        assertEquals(order1, order2);
    }
}
```

Example with ISP

The following example modifies the previous example to fix the violation. There are four different interfaces which are granular. The `OrderProvider` class is still single class implementing all four interfaces. `CachingReader` implementation now does not need to implement unnecessary methods.

UML representation




```
public interface ICreator<T> {
    void create(T item);
}
```

```
public interface IReader<T> {
    T read(int id);
}
```

```
public interface IUpdater<T> {
    void update(T item);
}
```

```
public interface IDeleter<T> {
    void delete(T item);
}
```

```
public class CachingReader<T> implements IReader<T>{
    private IReader<T> reader;
    private Map<Integer, T> itemMap = new HashMap<>();

    public CachingReader(IReader<T> reader) {
        this.reader = reader;
    }

    public T read(int id) {
        T item = itemMap.get(id);
        if( item == null ) {
```

```
            item = reader.read(id);
            itemMap.put(id, item);
        }
        return item;
    }
}
```

```
public class OrderProvider<Order>
implements IReader<Order>, IUpdater<Order>,
    ICreator<Order>, IDeleter<Order> {
    public void create(Order item) {
        // create logic
    }

    public Order read(int id) {
        return (Order)(new com.company.example.Order(id));
    }

    public void update(Order item) {
        // update logic
    }

    public void delete(Order item) {
        // delete logic
    }
}
```

```
public class OrderController {
    private IReader<Order> reader;
    private ICreator<Order> creator;
    private IUpdater<Order> updater;
    private IDeleter<Order> deleter;

    public OrderController(
        IReader<Order> reader,
        ICreator<Order> creator,
        IUpdater<Order> updater,
        IDeleter<Order> deleter
    ) {
        this.reader = new CachingReader(reader);
        this.creator = creator;
        this.updater = updater;
        this.deleter = deleter;
    }

    public Order getOrder(int id) {
        return reader.read(id);
    }
```

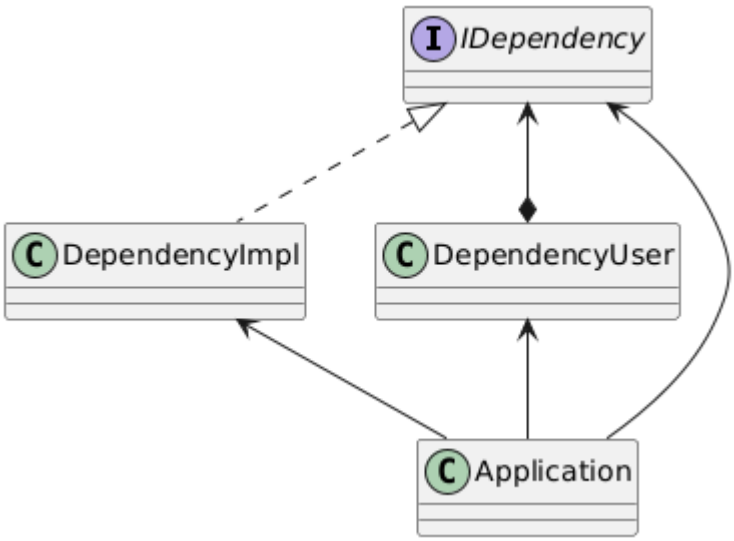
```
}
```

```
class OrderControllerTest {

    @Test
    void test_getOrder() {
        OrderProvider provider = new OrderProvider();
        OrderController orderController = new OrderController(
            provider,
            provider,
            provider,
            provider
        );

        Order order1 = orderController.getOrder(1);
        assertEquals(1, order1.getId());

        Order order2 = orderController.getOrder(1);
        assertEquals(1, order1.getId());

        assertEquals(order1, order2);
    }
}
```

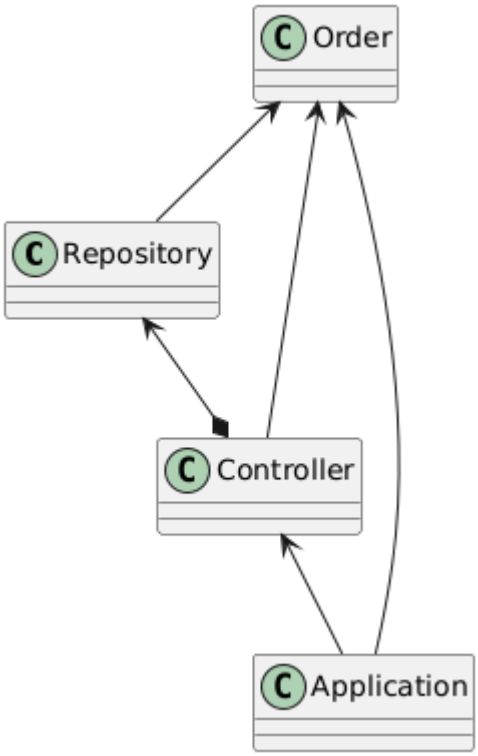
Dependency Inversion Principle

Dependency users should not create dependencies, but should reference the dependencies as interface references. The dependencies shall be injected by the calling context. This principle makes the objects loosely coupled and testable.

Basic Components



Example Without DI




```
public class Order {
    private int id;

    public Order(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

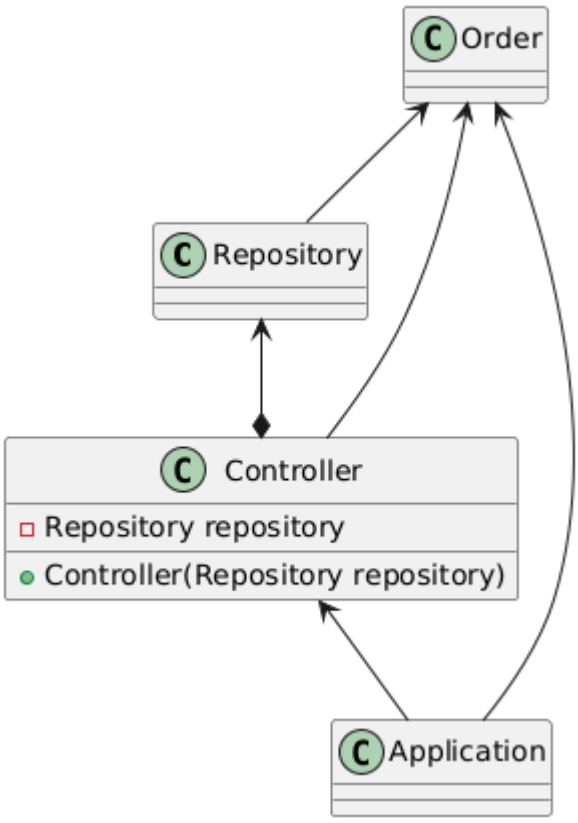
```
public class Repository {
    public void save(Order order) {
        // save logic
    }
}
```

```
public class Controller {
    Repository repository = new Repository(); ①

    public void save(Order order) {
        repository.save(order);
    }
}
```

① Repository object created here binds this class with the dependency

```
class ControllerTest {

    @Test
    void test_save() {
        Controller controller = new Controller();
        Order order = new Order(100);
        controller.save(order); // no way to verify with mock
    }

    private class MockRepository extends Repository {
        private Order order;

        public void save(Order order) {
            this.order = order;
        }

        public int getId() {
            return order.getId();
        }
    }
}
```

Example With DI




```
public class Order {
```

```
    private int id;

    public Order(int id) {
        this.id = id;
    }

    public int getId() {
        return id;
    }
}
```

```
public class Repository {
    public void save(Order order) {
        // save logic
    }
}
```

```
public class Controller {
    Repository repository;

    public Controller(Repository repository) { ①
        this.repository = repository;
    }

    public void save(Order order) {
        repository.save(order);
    }
}
```

① Controller is expecting dependency to be injected here

```
class ControllerTest {

    @Test
    void test_save() {
        MockRepository repository = new MockRepository();
        Controller controller = new Controller(repository); ①
        Order order = new Order(100);
        controller.save(order);

        assertEquals(100, repository.getId());
    }

    private class MockRepository extends Repository {
        private Order order;

        public void save(Order order) {
            this.order = order;
```

```
        }

        public int getId() {
            return order.getId();
        }
    }
}
```

① Dependency is injected like this by the Application

Summary

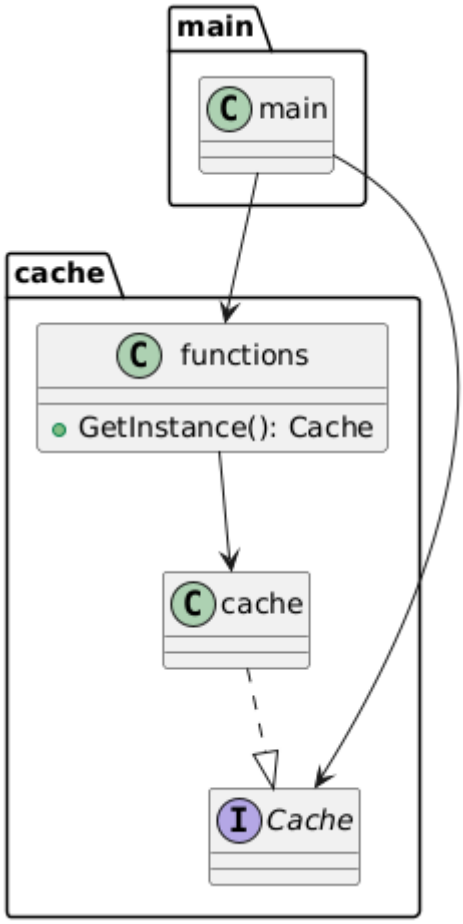
While applying design patterns to software design keeping the SOLID principles in mind will make the design robust. SOLID principles help in writing clean code which is maintainable, readable, testable and extensible.

Singleton Pattern

GoF Definition

Ensure a class only has one instance, and provide a global point of access to it.

UML Representation



Eager Initialization

.Cache.java

```
package com.example;

public final class Cache {
    private static final Cache INSTANCE = new Cache();

    private Cache() {
        // avoid initialization
    }

    public static Cache getInstance() {
        return INSTANCE;
```

```
    }
}
```

Lazy Loading with thread safety

.Cache.java

```
package com.example;

public final class Cache {
    private static Cache cache;

    private Cache() {
        // avoid initialization
    }

    public static synchronized Cache getInstance() {
        if( cache == null) {
            cache = new Cache();
        }
        return cache;
    }
}
```

Lazy Loading with Double Checked Locking

.Cache.java

```
package com.example;

public final class Cache {
    private static Cache cache;

    private Cache() {
        // avoid initialization
    }

    public static Cache getInstance() {
        if( cache == null) {
            synchronized(Cache.class) {
                if( cache == null) {
                    cache = new Cache();
                }
            }
        }
        return cache;
    }
```

```
}
```

Bill Pugh's Static Helper

.Cache.java

```
package com.example;

public final class Cache {
    private static class SingletonHelper {
        private static final Cache INSTANCE = new Cache();
    }

    private Cache() {
        // avoid initialization
    }

    public static Cache getInstance() {
        return SingletonHelper.INSTANCE;
    }
}
```

Interface Based Implementation

.Cache.java

```
package com.example;

import java.util.Map;
import java.util.HashMap;

public final class Cache implements ICache {
    private static final Cache INSTANCE = new Cache();

    private final Map<String, String> cacheMap = new HashMap<>();

    private Cache() {
        // avoid initialization
    }

    public static ICache getInstance() {
        return INSTANCE;
    }

    public String getValue(String key) {
        return cacheMap.get(key);
    }
```

```
    public void setValue(String key, String value) {
        cacheMap.put(key, value);
    }
}
```

Spring Boot Version

Cache.java

```
package com.example;

import org.springframework.stereotype.Component;

@Component
public class Cache {

}
```

AcceptList.java

```
package com.example;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class AcceptList {

    @Autowired
    Cache cache;

    public Cache getCache() {
        return cache;
    }
}
```

Singleton and Enum by Joshua Bloch

```
package com.journaldev.singleton;

public enum Cache {

    INSTANCE;
```

```
    public static void doSomething(){
        //do something
    }
}
```

Singleton and reflection

Singleton objects can be instantiated using reflection and hence the objective can be overridden using reflection

Singleton and serialization

If the singleton class supports serialization then it can be serialized and deserialized to override the objective of maintaining one instance.

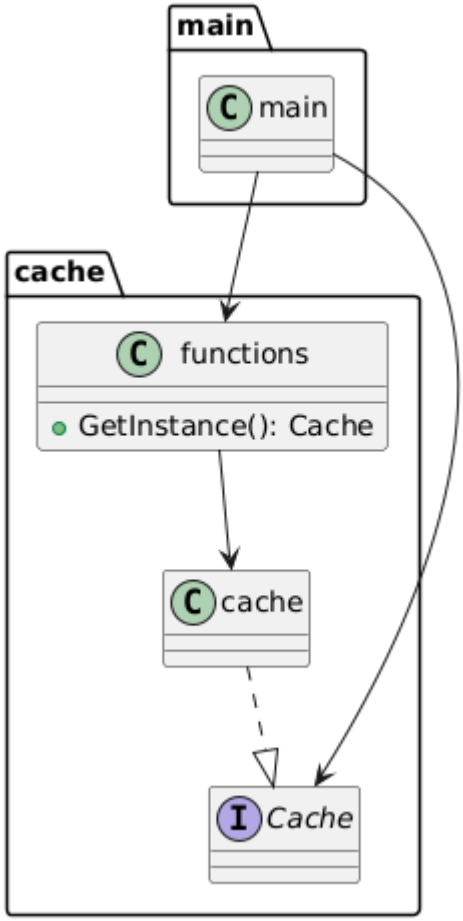
Singleton Instances in JDK

Singleton examples in Java core libraries:

- java.lang.Runtime#getRuntime()
- java.awt.Desktop#getDesktop()
- java.lang.System#getSecurityManager()

Singleton example in Go

TBD




```go
package cache

import (
    "fmt"
    "sync"

    "github.com/rs/zerolog/log"
)

// Cache is reference to a cache instance.
// A Cache is safe for concurrent use by multiple goroutines.
type Cache[T any] interface {
    // Get returns value stored for key and exists=true.
    //
    // When value is not found exists=false is returned.
    Get(key string) (value T, exists bool)

    // Set stores the value for given key
    Set(key string, value T)
}

type cache[T any] struct {
    mutex sync.RWMutex
    items map[string]interface{}
}

```

```
var (
    instance Cache[any]
    once     sync.Once
)

func createCacheInstance[T any]() {
    instance = &cache[any]{
        items: make(map[string]interface{}),
    }
    log.Info().Msg(fmt.Sprintf("Cache instance created for type %T", *new(T)))
}

// GetInstance returns singleton instance of Cache
func GetInstance[T any]() Cache[any] {
    once.Do(createCacheInstance[T])
    return instance
}

func (c *cache[T]) Set(key string, value T) {
    c.mutex.Lock()
    defer c.mutex.Unlock()
    c.items[key] = value
}

func (c *cache[T]) Get(key string) (value T, exists bool) {
    c.mutex.RLock()
    defer c.mutex.RUnlock()
    rawValue, exists := c.items[key]
    if exists {
        value = rawValue.(T)
    }
    return value, exists
}
```

```
package main

import (
    "fmt"
    "singleton/cache"
    "sync"

    "github.com/rs/zerolog/log"
)

func process(index int) {
    cache := cache.GetInstance[string]()
    cache.Set("one", fmt.Sprintf("one:%d", index))
    cache.Set("two", fmt.Sprintf("two:%d", index))

    value, exist := cache.Get("one")
```

```
17      if exist {
18          log.Info().Msg(fmt.Sprintf("Process: %d, Value: %s", index, value))
19      }
20
21      value2, exist := cache.Get("two")
22      if exist {
23          log.Info().Msg(fmt.Sprintf("Process: %d, Value: %s", index, value2))
24      }
25  }
26
27  func main() {
28      waitGroup := sync.WaitGroup{}
29      for i := 0; i < 5; i++ {
30          waitGroup.Add(1)
31          go func() {
32              defer waitGroup.Done()
33              process(i)
34          }()
35      }
36      waitGroup.Wait()
37  }
```

Factory Pattern

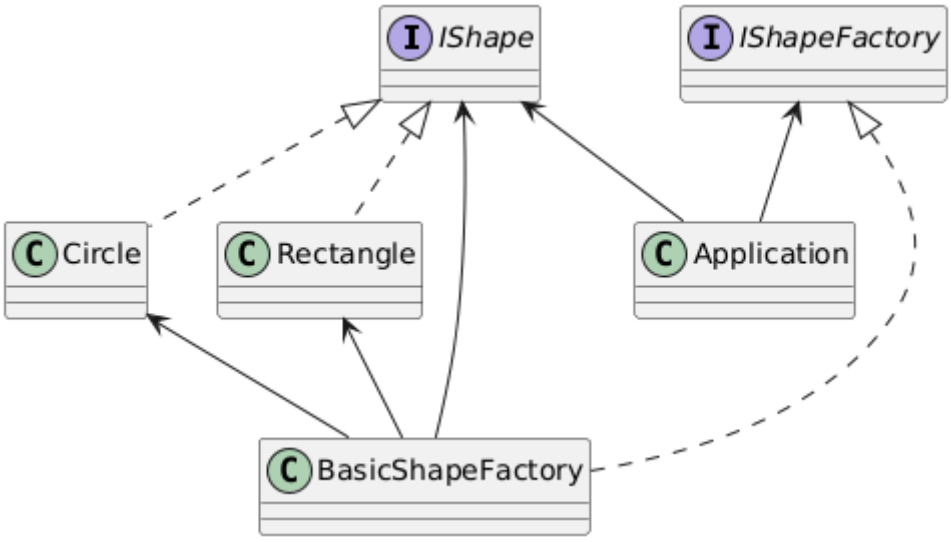
GoF Definition

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Things to note about Factory Pattern

- This pattern caters to creation of objects of specific interface/class, let us call it `Product`
- The term interface is referred as a generic concept that is a contract expressed for the calling code to expect in the `Product`
- Factory or Creator class is used to create the objects of the specific interface
- The concrete implementation which is returned from the `Factory` method is not known to the calling code

UML Interpretation



Basic Factory

```
public interface IShape {
    String getName();
}
```

```
public interface IShapeFactory {
    IShape create(String type);
```

```
}
```

```
public class Circle implements IShape {
    public String getName() {
        return "Circle";
    }
}
```

```
public class Rectangle implements IShape {
    public String getName() {
        return "Rectangle";
    }
}
```

```
public class BasicShapeFactory implements IShapeFactory {
    public IShape create(String type) {
        if( "CIRCLE".equals(type)) {
            return new Circle();
        } else if( "RECTANGLE".equals(type)) {
            return new Rectangle();
        } else {
            return null;
        }
    }
}
```

```
class BasicShapeFactoryTest {

    private BasicShapeFactory target;

    @BeforeEach
    public void beforeEach() {
        target = new BasicShapeFactory();
    }

    @Test
    void check_object_created() {
        IShape circle = target.create("CIRCLE");
        IShape rectangle = target.create("RECTANGLE");

        assertEquals("Circle", circle.getName());
        assertEquals("Rectangle", rectangle.getName());
    }

    @Test
    void check_object_are_not_same() {
```

```
        IShape circle1 = target.create("CIRCLE");
        IShape circle2 = target.create("CIRCLE");

        assertNotEquals(circle1, circle2);
    }

    @Test
    void check_invalid_type_returns_null() {
        IShape unknown = target.create("UNKNOWN");

        assertNull(unknown);
    }

}
```

Loose Coupling

Factory pattern can help achieve loose coupling between components.

`IAuditor` interface declares the expectation contract of an Auditor object.

```
public interface IAuditor {
    void log(String message);
}
```

The `AuditorFactory` class decides which concrete Auditor class to instantiate. This separates the decision of which Auditor to use from the actual calling components.

The decision of choosing a specific auditor can be externalized as a property declaration or even an environmental variable declaration in the `AuditorFactory` class.

```
public class AuditorFactory {
    public static IAuditor create() {
        return new FileSystemAuditor();
    }
}
```

Encapsulation using Factory

Factory can encapsulate complex logic of building the service. The logic of creating additional dependencies and building the `IClaimCalculator` is encapsulated in the `ClaimCalculatorFactory`

```
public class ClaimCalculatorFactory {
    public static IClaimCalculator create() {
        RuleList ruleList = new RuleList();
        FixedConditions fixedConditions =  new FixedConditions();
```

```
        return new StandardClaimCalculator(ruleList, fixedConditions);
    }
}
```

```
public class ClaimCreator {
    public double create(String code, double amount) {
        IClaimCalculator claimCalculator = ClaimCalculatorFactory.create();
        return claimCalculator.calculateClaimAmount(code, amount);
    }
}
```

Factory instances in JDK

Factory examples in Java core libraries:

- java.util.Calendar#getInstance()
- java.util.ResourceBundle#getBundle()
- java.text.NumberFormat#getInstance()
- java.nio.charset.Charset#forName()
- java.net.URLStreamHandlerFactory#createURLStreamHandler(String)
- java.util.EnumSet#of()
- javax.xml.bind.JAXBContext#createMarshaller()

Abstract Factory Pattern

GoF Definition

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Block Diagram

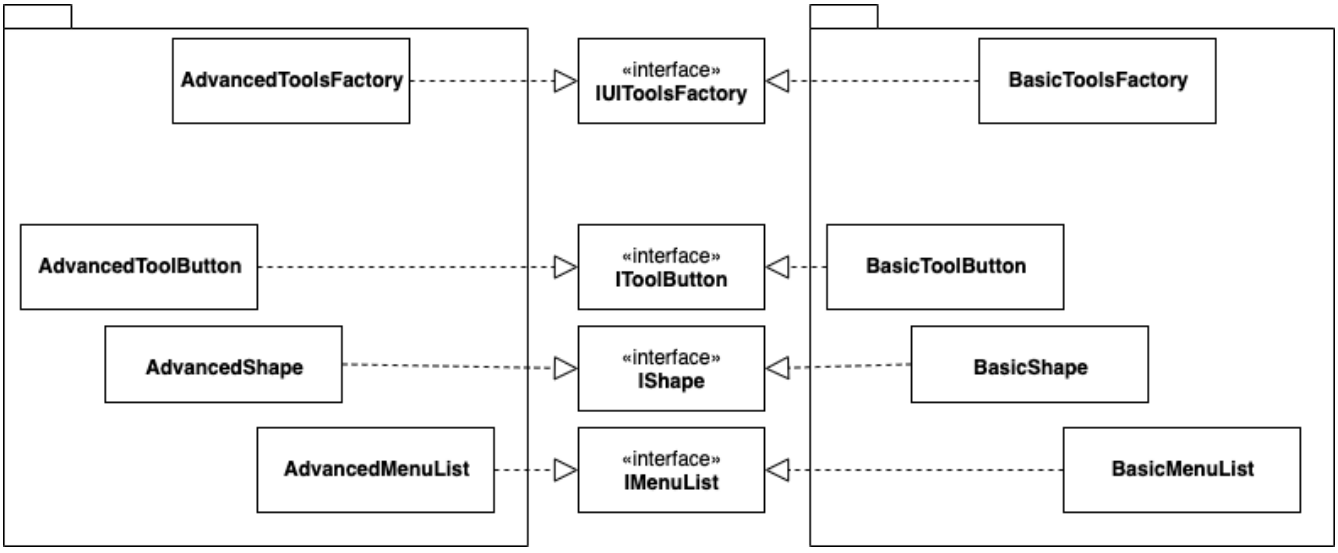
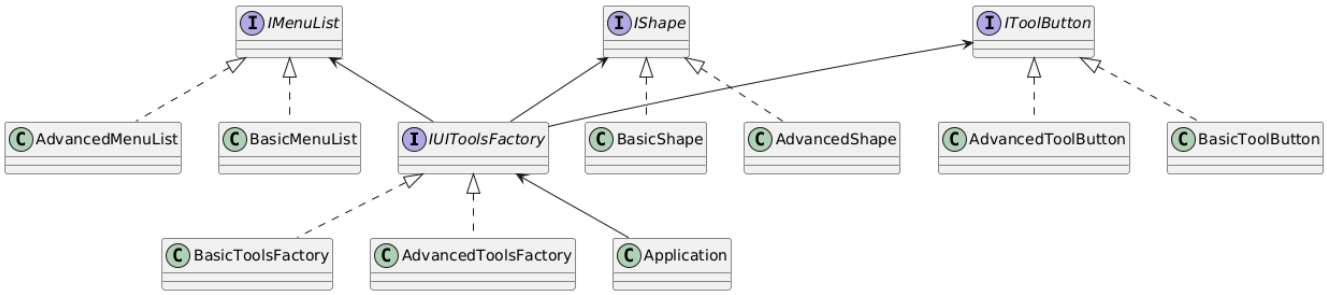


Figure 1. Image Components of Abstract Factory Pattern

UML Representation



UI Tools Example

`IUIToolsFactory` is public contract for any factory to satisfy. Any class implementing this interface agrees to provide a factory for three product interfaces `IShape`, `IToolButton` and `IMenuList`.

A set of Factory class and concrete product classes is created as one option for the factory implementation.

Similarly another Factory implementation is made available which can be replaced in the constructor of `Application` class to switch the products from one set of or family of products to another family of products.

```
public class Application {
    private final IUIToolsFactory toolsFactory;
    private final IToolButton toolButton;
    private final IShape shape;
    private final IMenuList menuList;

    public Application(IUIToolsFactory toolsFactory) {
        this.toolsFactory = toolsFactory;
        this.toolButton = toolsFactory.createToolButton("TOOL_1");
        this.shape = toolsFactory.createShape("CIRCLE");
        this.menuList = toolsFactory.createMenuList("FILE");
    }

    public IShape getShape() {
        return this.shape;
    }

    public IToolButton getToolButton() {
        return this.toolButton;
    }

    public IMenuList getMenuList() {
        return this.menuList;
    }
}
```

```
public interface IUIToolsFactory {
    IToolButton createToolButton(String type);
    IShape createShape(String type);
    IMenuList createMenuList(String type);
}
```

```
public interface IMenuList {
    public String getName();
}
```

```
public interface IToolButton {
    public String getName();
}
```

```
public interface IShape {
    public String getName();
}
```

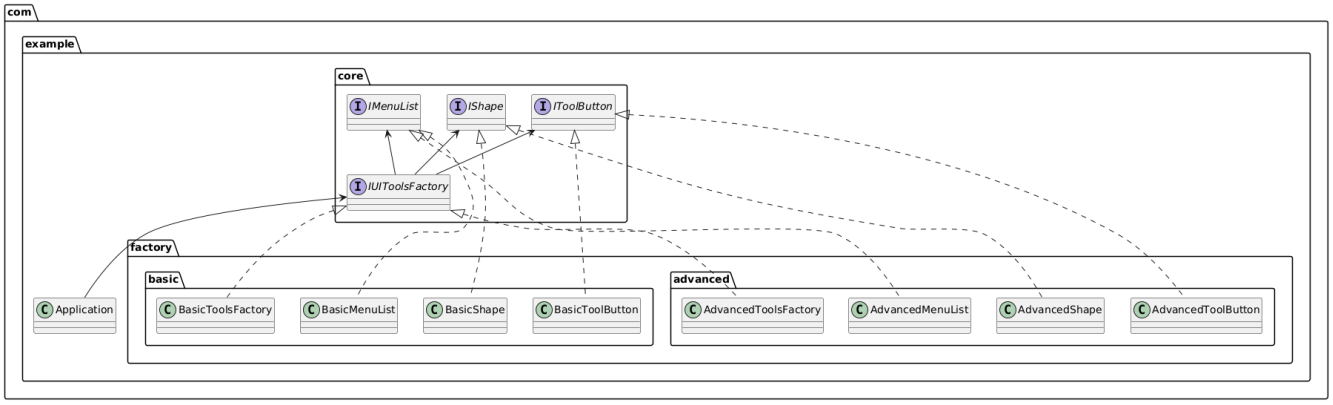
```
public class BasicToolsFactory implements IUIToolsFactory {
    public IToolButton createToolButton(String type) {
        return new BasicToolButton();
    }

    public IShape createShape(String type) {
        return new BasicShape();
    }

    public IMenuList createMenuList(String type) {
        return new BasicMenuList();
    }
}
```

```
public class AdvancedToolsFactory implements IUIToolsFactory {
    public IToolButton createToolButton(String type) {
        return new AdvancedToolButton();
    }

    public IShape createShape(String type) {
        return new AdvancedShape();
    }

    public IMenuList createMenuList(String type) {
        return new AdvancedMenuList();
    }
}
```

```
class ApplicationTest {

    private Application application;

    @Test
    void check_basic_tools_factory() {
        IUIToolsFactory toolsFactory = new BasicToolsFactory();
        application = new Application(toolsFactory);

        assertEquals("BasicShape", application.getShape().getName());
        assertEquals("BasicToolButton", application.getToolButton().getName());
        assertEquals("BasicMenuList", application.getMenuList().getName());
    }

    @Test
    void check_advanced_tools_factory() {
        IUIToolsFactory toolsFactory = new AdvancedToolsFactory();
        application = new Application(toolsFactory);

        assertEquals("AdvancedShape", application.getShape().getName());
```

```
        assertEquals("AdvancedToolButton", application.getToolButton().getName());
        assertEquals("AdvancedMenuList", application.getMenuList().getName());
    }

}
```

Separated Interface.[1]



Abstract Factory instances in JDK

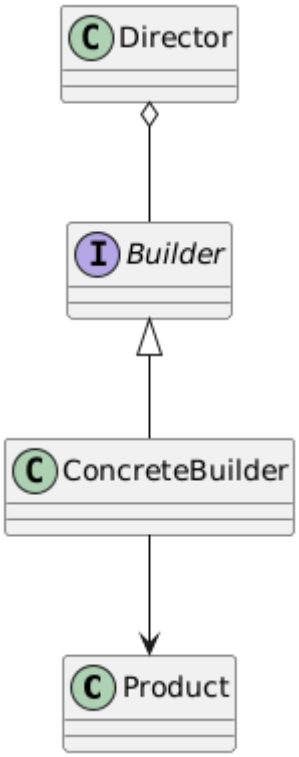
- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

Builder Pattern

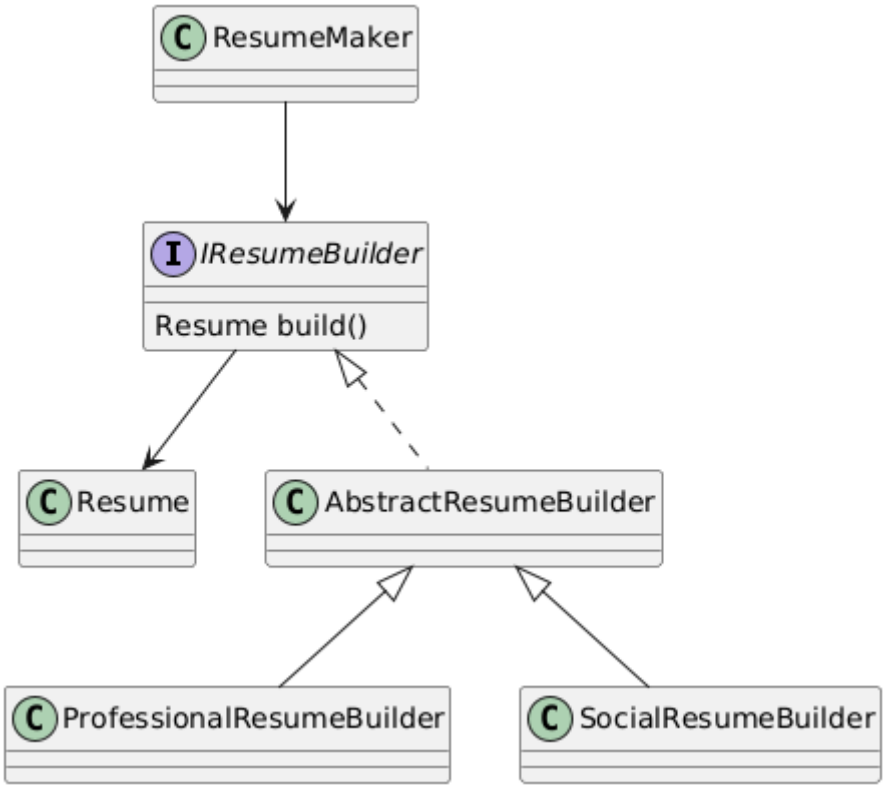
GoF Definition

Separate the construction of a complex object from its representation so that the same construction processes can create different representations.

UML Interpretation



Resume Builder Example




```
public interface IResumeBuilder {
    void createEducationDetails();
    void createProjectDetails();
    void createSocialDetails();
    void createTestimonialDetails();
    void createPersonalDetails();
    void createCertificationDetails();
    void createNameAndTitle();
    Resume build();
}
```

```
public class Resume {
    private List<String> sectionList = new ArrayList<>();

    public void addSection(String section) {
        sectionList.add(section);
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        for(String section : sectionList) {
            builder.append(section);
        }
        return builder.toString();
    }
}
```

```
public abstract class AbstractResumeBuilder implements IResumeBuilder {

    Resume resume = new Resume();

    public Resume build() {
        Resume resumeToReturn = resume;
        resume = new Resume();
        return resumeToReturn;
    }
}
```

```
public class ResumeMaker {

    private IResumeBuilder builder;

    public ResumeMaker(IResumeBuilder builder) {
        this.builder = builder;
    }

    public void makeCV() {
        builder.createNameAndTitle();
        builder.createProjectDetails();
        builder.createTestimonialDetails();
        builder.createCertificationDetails();
        builder.createEducationDetails();
        builder.createSocialDetails();
        builder.createPersonalDetails();
    }

    public void makeResume() {
        builder.createNameAndTitle();
        builder.createProjectDetails();
        builder.createCertificationDetails();
        builder.createEducationDetails();
        builder.createPersonalDetails();
    }

    public void makeOnePagerResume() {
        builder.createNameAndTitle();
        builder.createProjectDetails();
        builder.createPersonalDetails();
    }
}
```

```
public class ProfessionalResumeBuilder extends AbstractResumeBuilder {

    public void createEducationDetails() {
        resume.addSection("Professional Education");
```

```
    }

    public void createProjectDetails() {
        resume.addSection("Professional Projects");
    }

    public void createSocialDetails() {
        resume.addSection("Professional Social");
    }

    public void createTestimonialDetails() {
        resume.addSection("Professional Testimonials");
    }

    public void createPersonalDetails() {
        resume.addSection("Professional Personal");
    }

    public void createCertificationDetails() {
        resume.addSection("Professional Certifications");
    }

    public void createNameAndTitle() {
        resume.addSection("Professional NameAndTitle");
    }
}
```

```
public class SocialResumeBuilder extends AbstractResumeBuilder {

    public void createEducationDetails() {
        resume.addSection("Social Education");
    }

    public void createProjectDetails() {
        resume.addSection("Social Projects");
    }

    public void createSocialDetails() {
        resume.addSection("Social Social");
    }

    public void createTestimonialDetails() {
        resume.addSection("Social Testimonials");
    }

    public void createPersonalDetails() {
        resume.addSection("Social Personal");
    }

    public void createCertificationDetails() {
```

```
        resume.addSection("Social Certifications");
    }

    public void createNameAndTitle() {
        resume.addSection("Social NameAndTitle");
    }
}
```

```
class ResumeMakerTest {

    @Test
    public void ensure_professional_cv() {
        IResumeBuilder builder = new ProfessionalResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeCV();
        Resume resume = builder.build();

        String cvText = "Professional NameAndTitle"
                        + "Professional Projects"
                        + "Professional Testimonials"
                        + "Professional Certifications"
                        + "Professional Education"
                        + "Professional Social"
                        + "Professional Personal";

        assertEquals(cvText, resume.toString());
    }

    @Test
    public void ensure_professional_resume() {
        IResumeBuilder builder = new ProfessionalResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeResume();
        Resume resume = builder.build();

        String cvText = "Professional NameAndTitle"
                        + "Professional Projects"
                        + "Professional Certifications"
                        + "Professional Education"
                        + "Professional Personal";

        assertEquals(cvText, resume.toString());
    }

    @Test
    public void ensure_professional_one_pager() {
        IResumeBuilder builder = new ProfessionalResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeOnePagerResume();
        Resume resume = builder.build();
```

```
        String cvText = "Professional NameAndTitle"
                        + "Professional Projects"
                        + "Professional Personal";

        assertEquals(cvText, resume.toString());
    }

    @Test
    public void ensure_social_cv() {
        IResumeBuilder builder = new SocialResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeCV();
        Resume resume = builder.build();

        String cvText = "Social NameAndTitle"
                        + "Social Projects"
                        + "Social Testimonials"
                        + "Social Certifications"
                        + "Social Education"
                        + "Social Social"
                        + "Social Personal";

        assertEquals(cvText, resume.toString());
    }

    @Test
    public void ensure_social_resume() {
        IResumeBuilder builder = new SocialResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeResume();
        Resume resume = builder.build();

        String cvText = "Social NameAndTitle"
                        + "Social Projects"
                        + "Social Certifications"
                        + "Social Education"
                        + "Social Personal";

        assertEquals(cvText, resume.toString());
    }

    @Test
    public void ensure_social_one_pager() {
        IResumeBuilder builder = new SocialResumeBuilder();
        ResumeMaker maker = new ResumeMaker(builder);
        maker.makeOnePagerResume();
        Resume resume = builder.build();

        String cvText = "Social NameAndTitle"
                        + "Social Projects"
```

```
                        + "Social Personal";

        assertEquals(cvText, resume.toString());
    }
}
```

Builder instances in JDK

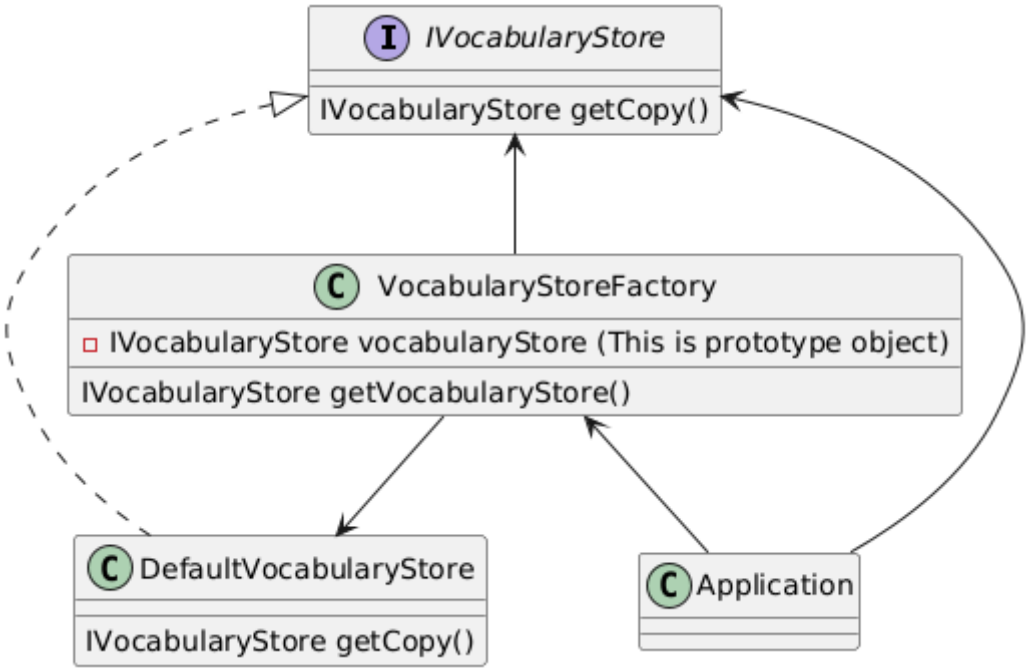
- java.lang.StringBuilder#append()
- java.lang.StringBuffer#append()
- java.nio.ByteBuffer#put()

Prototype Pattern

GoF Definition

Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

UML Representation



Vocabulary Store Example

```
public interface IVocabularyStore {
    int getWordCount();
    Set<String> getWordList();
    String getMeaning(String word);
    List<String> getUnlearnedWords();
    void setWordLearned(String word);
    void setWordUnLearned(String word);
    IVocabularyStore getCopy();
}
```

```
public class DefaultVocabularyStore implements IVocabularyStore {

    private Map<String, String> wordMap;

    public DefaultVocabularyStore() {
```

```
        this.wordMap = new HashMap<>();
        // expensive word loading operation here
        this.wordMap.put("feature", "n. Not an issue");
        this.wordMap.put("night", "n. Slogging for the whole night");
    }

    private DefaultVocabularyStore(Map<String, String> wordMap) {
        this.wordMap = wordMap;
    }

    public int getWordCount() {
        return wordMap.size();
    }

    public Set<String> getWordList() {
        return wordMap.keySet();
    }

    public String getMeaning(String word) {
        return wordMap.get(word);
    }

    public List<String> getUnlearnedWords() {
        // some logic
        return null;
    }

    public void setWordLearned(String word) {
        // some logic
    }

    public void setWordUnLearned(String word) {
        // some logic
    }

    public IVocabularyStore getCopy() {
        Map<String, String> newWordMap = new HashMap<>();
        for(Entry<String, String> entry : wordMap.entrySet()) {
            newWordMap.put(entry.getKey(), entry.getValue());
        }
        return new DefaultVocabularyStore(newWordMap);
    }
}
```

```
public class VocabularyStoreFactory {
    private static IVocabularyStore vocabularyStore = new DefaultVocabularyStore();

    public static IVocabularyStore getVocabularyStore() {
        return vocabularyStore.getCopy();
    }
```

```
}
```

```
class VocabularyStoreFactoryTest {

    @Test
    void check_stores_are_different() {
        IVocabularyStore store1 = VocabularyStoreFactory.getVocabularyStore();
        IVocabularyStore store2 = VocabularyStoreFactory.getVocabularyStore();
        assertNotEquals(store1, store2);
    }

    @Test
    void check_stores_are_differentcontain_base_words() {
        IVocabularyStore store1 = VocabularyStoreFactory.getVocabularyStore();
        IVocabularyStore store2 = VocabularyStoreFactory.getVocabularyStore();

        assertEquals("n. Not an issue", store1.getMeaning("feature"));
        assertEquals("n. Slogging for the whole night", store1.getMeaning("night"));

        assertEquals("n. Not an issue", store2.getMeaning("feature"));
        assertEquals("n. Slogging for the whole night", store2.getMeaning("night"));
    }
}
```

Prototype instances in JDK

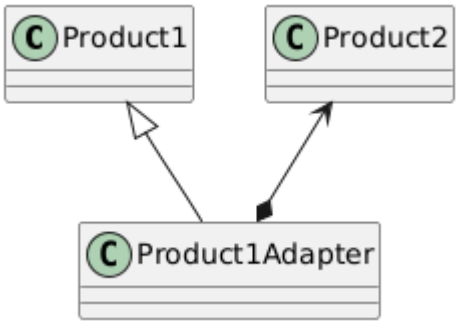
- Object#clone()

Adapter Pattern

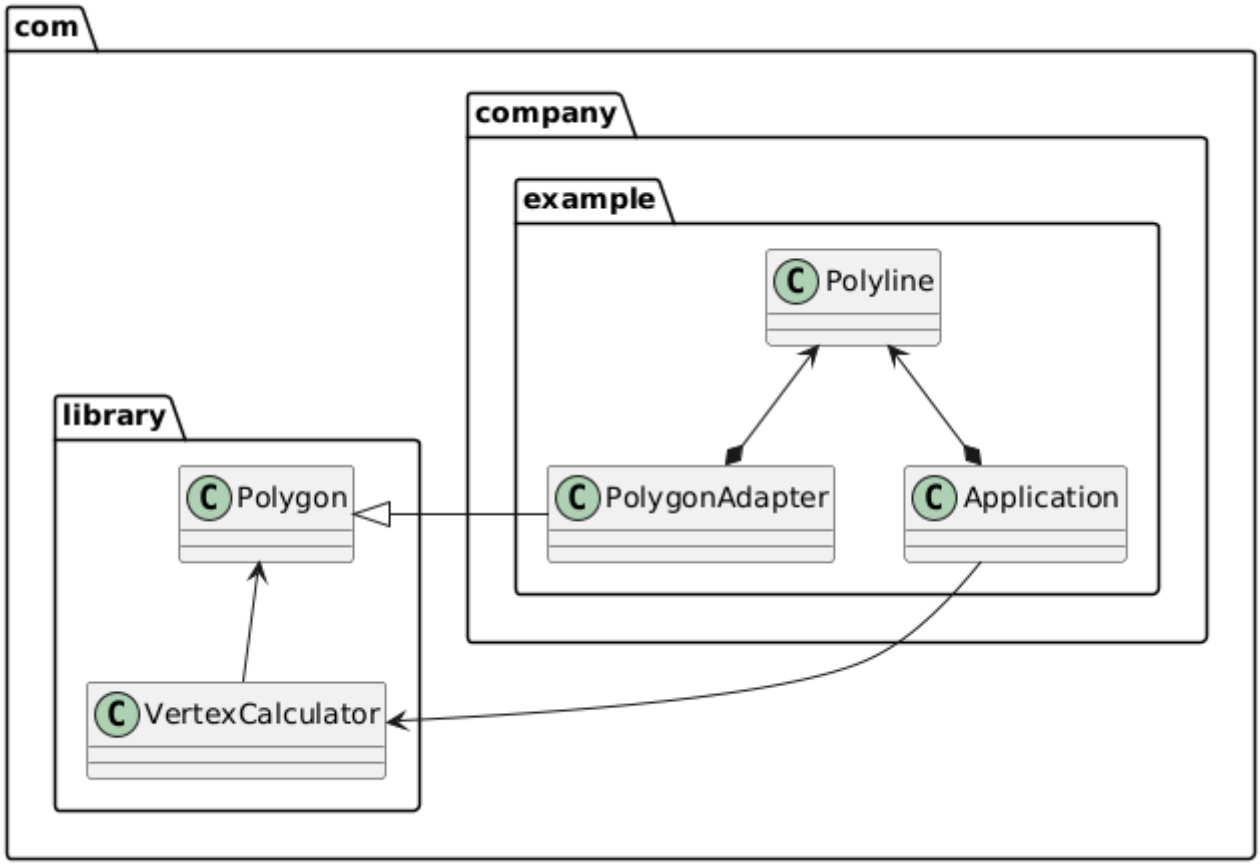
GoF Definition

Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

Basic Components



UML Representation




```
public class Polygon {
    public int getCount() {
        return 0;
    }
```

```
    public Point getCoordinate(int index) {
        return null;
    }
}
```

```
public class VertexCalculator {
    public static int countVertices(Polygon polygon) {
        return polygon.getCount();
    }
}
```

```
public class Polyline {
    private List<Coordinate> list = new ArrayList<>();

    public int getCount() {
        return list.size();
    }

    public Coordinate getCoordinate(int index) {
        return list.get(index);
    }

    public void addCoordinate(Coordinate coordinate) {
        list.add(coordinate);
    }
}
```

```
public class PolygonAdapter extends Polygon {
    private Polyline polyline;

    public PolygonAdapter(Polyline polyline) {
        this.polyline = polyline;
    }

    public int getCount() {
        return polyline.getCount();
    }

    public Point getCoordinate(int index) {
        Coordinate coordinate = polyline.getCoordinate(index);
        Point point = new Point();
        point.setX(coordinate.getLongitude());
        point.setY(coordinate.getLatitude());
        return point;
    }
```

```
}
```

```
public class Application {
    private Polyline polyline;

    public Application() {
        polyline = new Polyline();
        polyline.addCoordinate(new Coordinate(0,2));
        polyline.addCoordinate(new Coordinate(3,5));
        polyline.addCoordinate(new Coordinate(7,4));
    }

    public int calculateCornerCount() {
        return VertexCalculator.countVertices(new PolygonAdapter(polyline));
    }
}
```

```
class ApplicationTest {

    private Application target;

    @BeforeEach
    public void beforeEach() {
        target = new Application();
    }

    @Test
    void check_object_is_same() {
        assertEquals(3, target.calculateCornerCount());
    }
}
```

Adapter instances in JDK

- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer)

Bridge Pattern

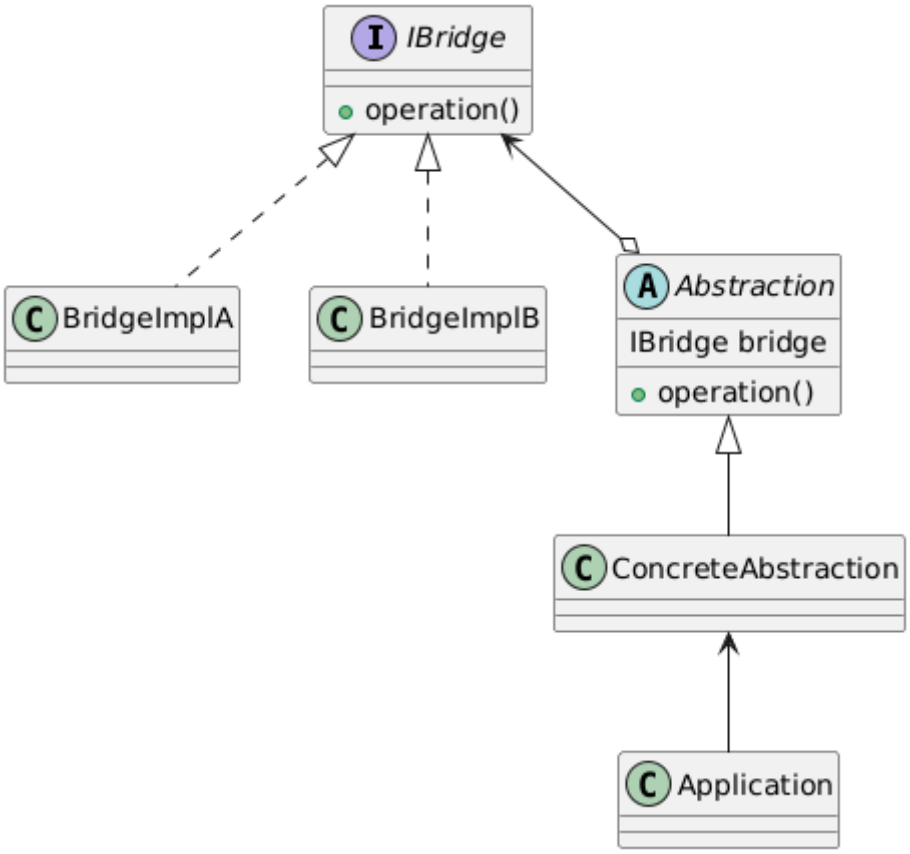
GoF Definition

Decouple an abstraction from its implementation so that the two can vary independently.

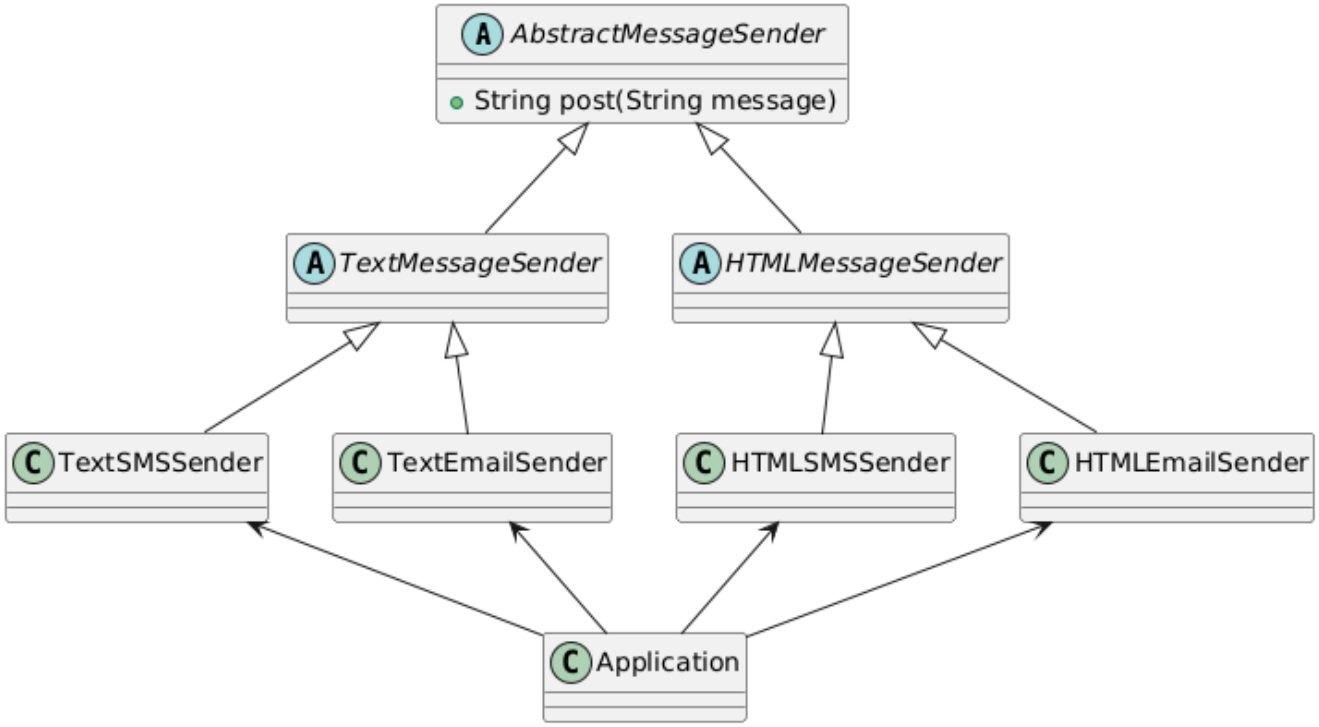
Basic Concept

Inheritance is a common way to specify different implementations of an abstraction. Using inheritance the implementations are bound tightly to the abstraction,and it is difficult to modify them independently. The Bridge pattern provides an alternative to inheritance when there is more than one version of an abstraction.

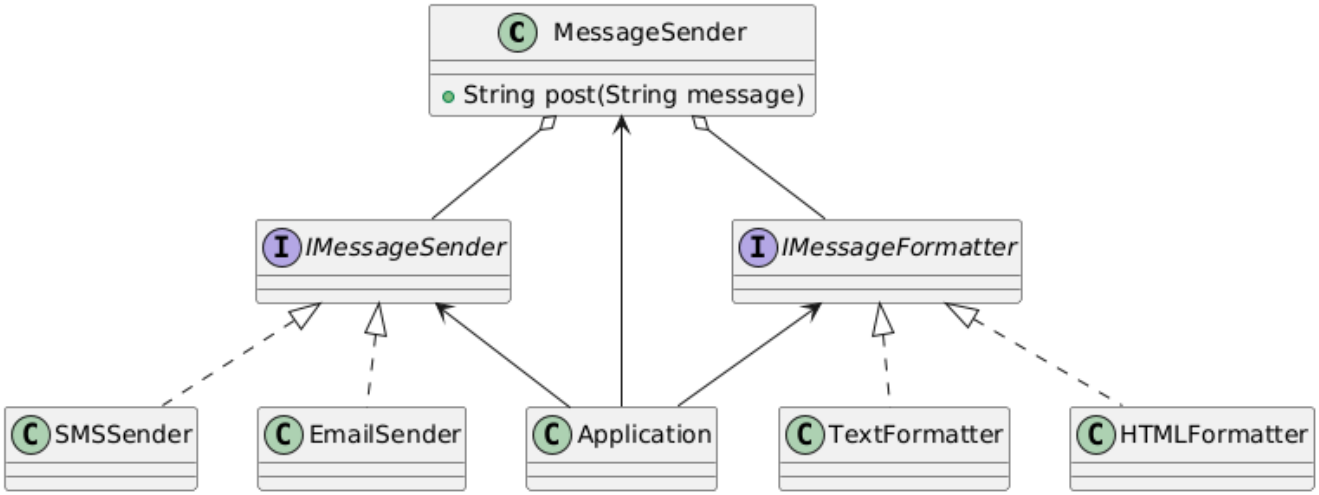
Components



Inheritance Example



Bridge Example




```
public interface IMessageSender {
    String sendMessage(String message);
}
```

```
public interface IMessageFormatter {
    String formatMessage(String message);
}
```

```
public class MessageSender {
    private IMessageSender sender;
    private IMessageFormatter formatter;
```

```
    public MessageSender(IMessageSender sender, IMessageFormatter formatter) {
        this.sender = sender;
        this.formatter = formatter;
    }

    public String post(String message) {
        String formattedMessage = formatter.formatMessage(message);
        return sender.sendMessage(formattedMessage);
    }
}
```

```
public class TextFormatter implements IMessageFormatter {
    public String formatMessage(String message) {
        return "Text Message: " + message;
    }
}
```

```
public class HTMLFormatter implements IMessageFormatter {
    public String formatMessage(String message) {
        return "HTML Message: " + message;
    }
}
```

```
public class SMSSender implements IMessageSender {
    public String sendMessage(String message) {
        return "SMS : " + message;
    }
}
```

```
public class EmailSender implements IMessageSender {
    public String sendMessage(String message) {
        return "Email : " + message;
    }
}
```

```
class MessageSenderTest {

    @Test
    void check_sms_text_sender() {
        IMessageFormatter formatter = new TextFormatter();
        IMessageSender sender = new SMSSender();
        MessageSender messageSender = new MessageSender(sender, formatter);
        String message = messageSender.post("Message");
```

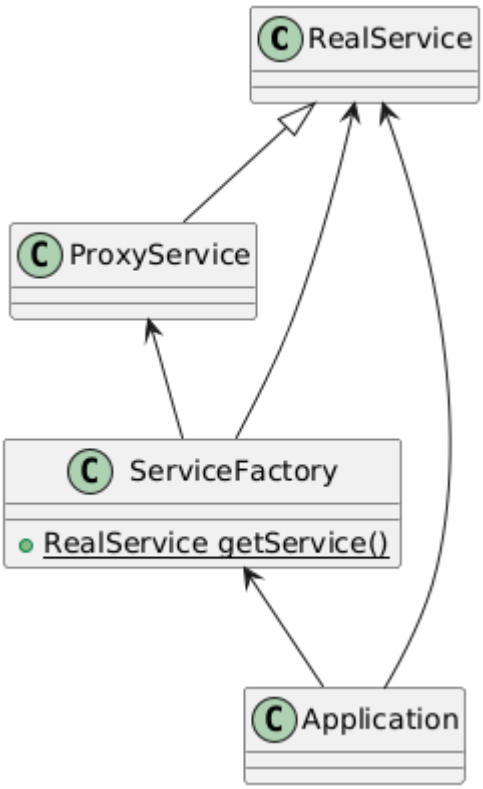
```
        assertEquals("SMS : Text Message: Message", message);
    }

    @Test
    void check_sms_html_sender() {
        IMessageFormatter formatter = new HTMLFormatter();
        IMessageSender sender = new SMSSender();
        MessageSender messageSender = new MessageSender(sender, formatter);
        String message = messageSender.post("Message");

        assertEquals("SMS : HTML Message: Message", message);
    }

    @Test
    void check_email_text_sender() {
        IMessageFormatter formatter = new TextFormatter();
        IMessageSender sender = new EmailSender();
        MessageSender messageSender = new MessageSender(sender, formatter);
        String message = messageSender.post("Message");

        assertEquals("Email : Text Message: Message", message);
    }

    @Test
    void check_email_html_sender() {
        IMessageFormatter formatter = new HTMLFormatter();
        IMessageSender sender = new EmailSender();
        MessageSender messageSender = new MessageSender(sender, formatter);
        String message = messageSender.post("Message");

        assertEquals("Email : HTML Message: Message", message);
    }
}
```

Proxy Pattern

GoF Definition

Provide a surrogate or placeholder for another object to control access to it.

Basic Components



Remote Proxies

Remote Proxies provide a local representation of another remote object or resource. Remote proxies are responsible not just for representation but also for some maintenance work. Such work could include connecting to a remote machine and maintaining the connection, encoding and decoding characters obtained through networking traffic, parsing, etc.

Virtual Proxies

Virtual Proxies wrap expensive objects and loads them on-demand. Sometimes we don't immediately need all functionalities that an object offers, especially if it is memory/time-consuming. Calling objects only when needed might increase performance quite a bit, as we'll see in the example below.

Protection Proxies

Protection Proxies are used for checking certain conditions. Some objects or resources might need

appropriate authorization for accessing them, so using a proxy is one of the ways in which such conditions can be checked. With protection proxies, we also get the flexibility of having many variations of access control.

Proxy instances in JDK

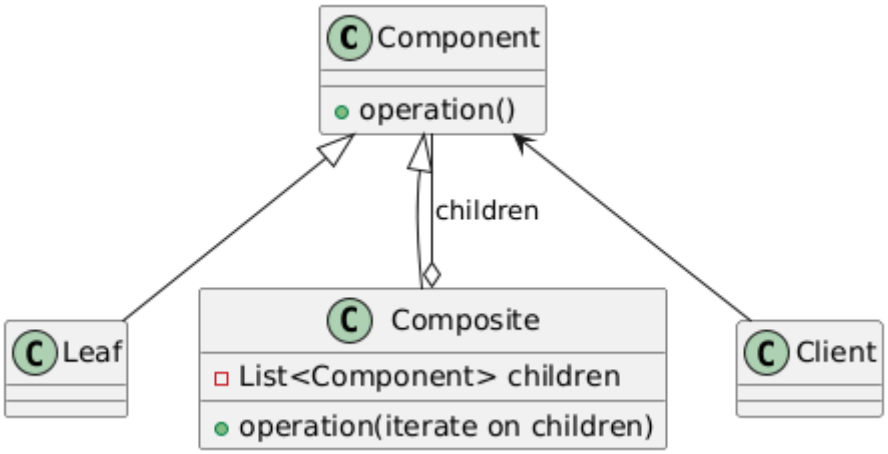
- java.lang.reflect.Proxy
- java.rmi.*
- javax.ejb.EJB (Proxies injected by framworks)
- javax.inject.Inject (Proxies injected by frameworks)
- javax.persistence.PersistenceContext

Composite Pattern

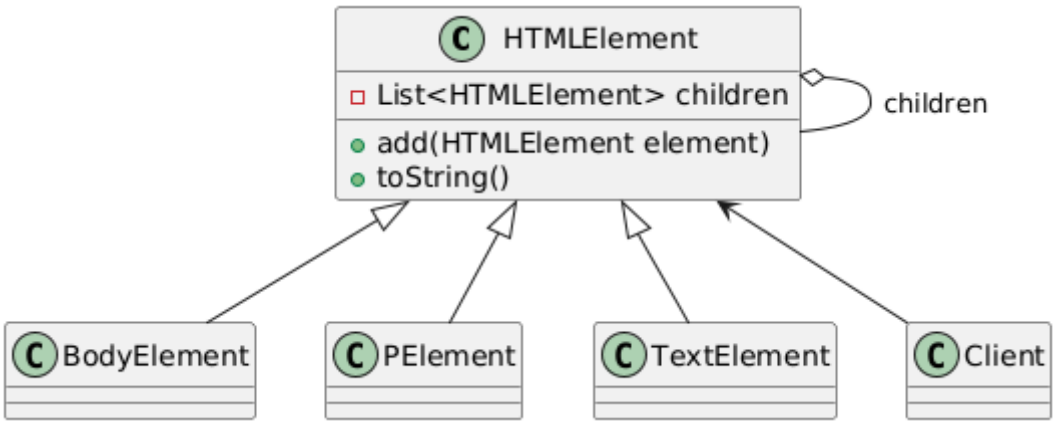
GoF Definition

Compose objects into tree structures to represent part-whole hierarchies. Com- posite lets clients treat individual objects and compositions of objects uniformly.

Basic Components



UML Representation




```
public class HTMLElement {
    String tagName;
    private List<HTMLElement> children = new ArrayList<>();

    public HTMLElement() {
        this("html");
    }

    public HTMLElement(String tagName) {
        this.tagName = tagName;
    }
```

```
    public void add(HTMLElement element) {
        children.add(element);
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("<");
        builder.append(tagName);
        builder.append(">");
        for(HTMLElement element : children) {
            builder.append(element.toString());
        }
        builder.append("</");
        builder.append(tagName);
        builder.append(">");

        return builder.toString();
    }
}
```

```
public class BodyElement extends HTMLElement {
    public BodyElement() {
        super("body");
    }
}
```

```
public class PElement extends HTMLElement {
    public PElement() {
        super("p");
    }
}
```

```
public class TextElement extends HTMLElement {
    public TextElement(String text) {
        super(text);
    }

    public String toString() {
        return tagName;
    }
}
```

```
class HTMLElementTest {

    @Test
    void test_html_element() {
```

```
        HTMLElement htmlElement = new HTMLElement();
        BodyElement body = new BodyElement();
        htmlElement.add(body);
        PElement p = new PElement();
        body.add(p);
        TextElement text = new TextElement("Hello World!");
        p.add(text);

        String expectedText = "<html>"
                              + "<body><p>Hello World!</p></body>"
                              + "</html>";

        assertEquals(expectedText, htmlElement.toString());
    }
}
```

Proxy instances in JDK

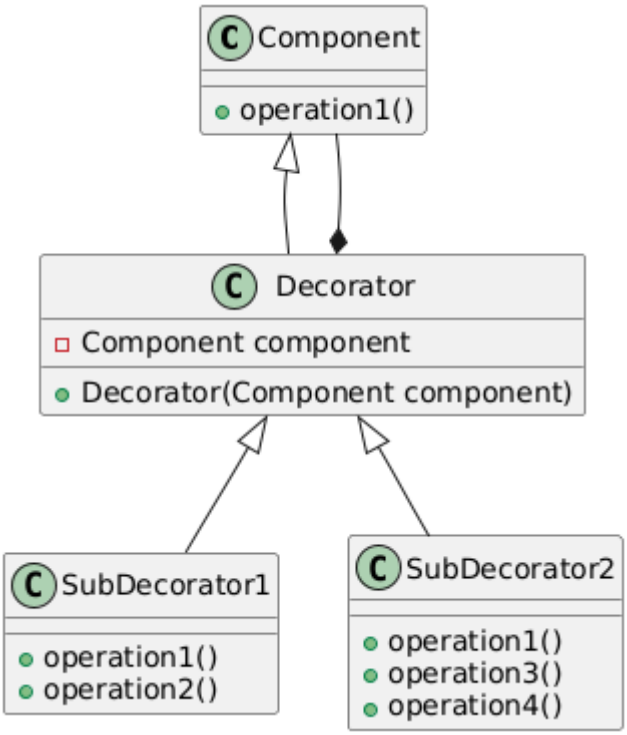
- java.awt.Container#add(Component)

Decorator Pattern

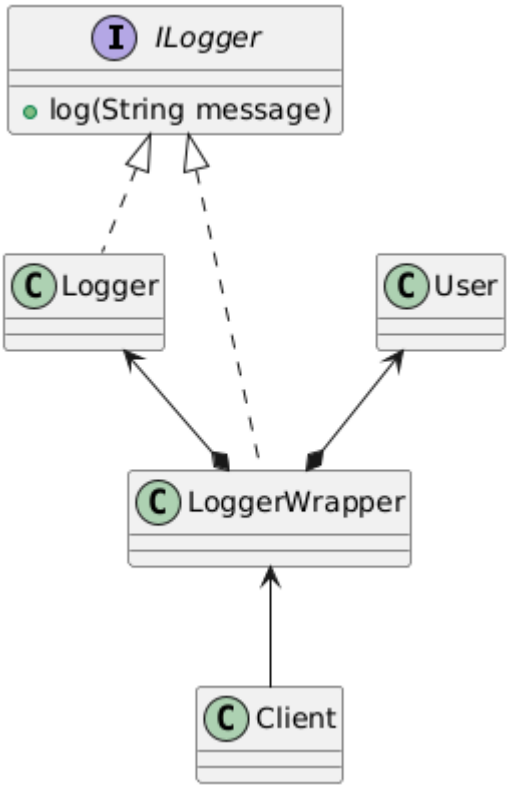
GoF Definition

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Basic Components



UML Representation




```
public interface ILogger {
    public String log(String message);
}
```

```
public class Logger implements ILogger {
    public String log(String message) {
        return message;
    }
}
```

```
public class User {
    private String name;

    public User(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

```
public class LoggerWrapper implements ILogger {
    private User user;
    private ILogger logger;
```

```
    public LoggerWrapper(ILogger logger) {
        this(logger, null);
    }

    public LoggerWrapper(ILogger logger, User user) {
        this.logger = logger;
        this.user = user;
    }

    public String log(User user, String message) {
        return logger.log("Called by " + user.getName() + " - " + message);
    }

    public String log(String message) {
        if( user != null ) {
            return logger.log("Called by " + user.getName() + " - " + message);
        } else {
            return logger.log(message);
        }
    }
}
```

```
class LoggerWrapperTest {

    @Test
    void test_no_user() {
        ILogger realLogger = new Logger();
        ILogger logger = new LoggerWrapper(realLogger);

        String value = logger.log("Some Message");
        assertEquals("Some Message", value);
    }

    @Test
    void test_user_passed_in_constructor() {
        Logger realLogger = new Logger();
        User user = new User("Tushar");
        ILogger logger = new LoggerWrapper(realLogger, user);

        String value = logger.log("Some Message");
        assertEquals("Called by Tushar - Some Message", value);
    }
}
```

Convenience Wrappers

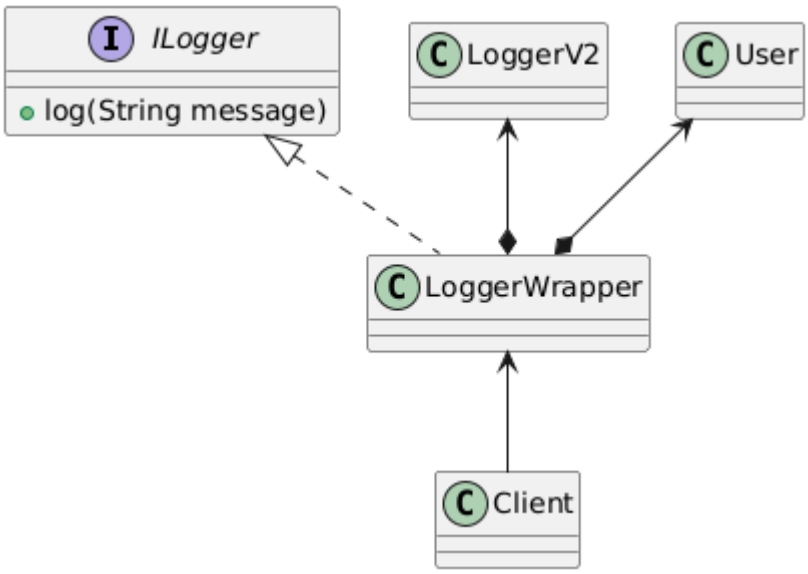
Decorator or Wrapper classes can be written for writing convenience methods over existing API

services and libraries. The operations needed in combination from the third party library can be created as an additional method of the wrapper.

Anti Corruption Layer

Wrapper classes can be written to abstract the third party service so it can be swapped with another service without impacting the rest of the project, as rest of the project only depends on the wrapper interface

UML Representation




```
public class LoggerV2 {
    public String logMessage(String message) {
        return message;
    }
}
```

```
public class LoggerWrapper implements ILogger {
    private User user;
    private LoggerV2 logger;

    public LoggerWrapper(LoggerV2 logger) {
        this(logger, null);
    }

    public LoggerWrapper(LoggerV2 logger, User user) {
        this.logger = logger;
        this.user = user;
    }

    public String log(User user, String message) {
        return logger.logMessage("Called by " + user.getName() + " - " + message);
```

```
    }

    public String log(String message) {
        if( user != null ) {
            return logger.logMessage("Called by " + user.getName() + " - " + message);
        } else {
            return logger.logMessage(message);
        }
    }
}
```

```
class LoggerWrapperTest {

    @Test
    void test_no_user() {
        LoggerV2 realLogger = new LoggerV2();
        ILogger logger = new LoggerWrapper(realLogger);

        String value = logger.log("Some Message");
        assertEquals("Some Message", value);
    }

    @Test
    void test_user_passed_in_constructor() {
        LoggerV2 realLogger = new Logger();
        User user = new User("Tushar");
        ILogger logger = new LoggerWrapper(realLogger, user);

        String value = logger.log("Some Message");
        assertEquals("Called by Tushar - Some Message", value);
    }
}
```

Decorator instances in JDK

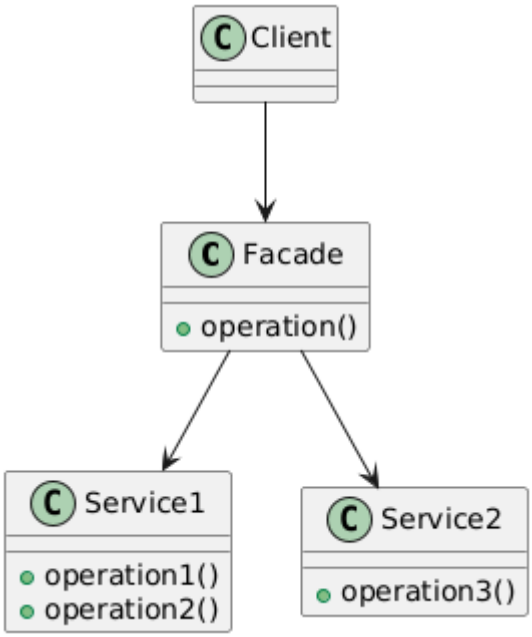
- java.io.InputStream
- java.io.InputStreamReader
- java.io.BufferedReader

Facade Pattern

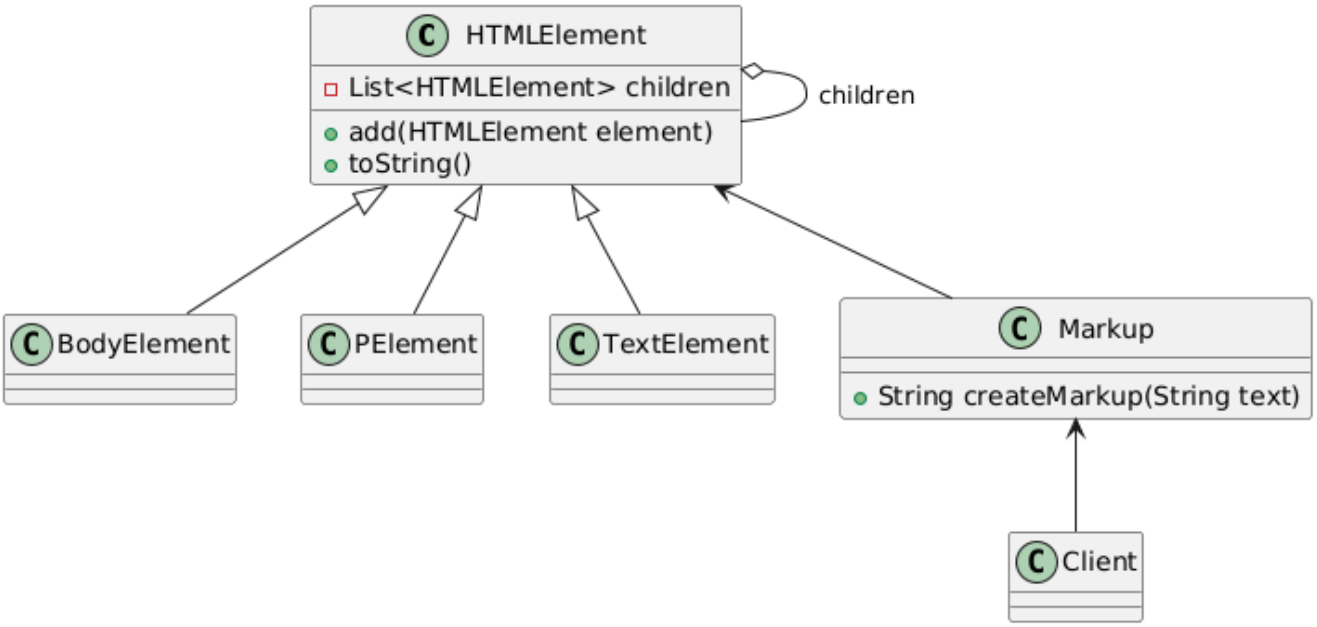
GoF Definition

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Basic Components



UML Representation




```
public class Markup {
    public String createMarkup(String text) {
```

```
        HTMLElement html = new HTMLElement();
        BodyElement body = new BodyElement();
        PElement p = new PElement();
        TextElement textElement = new TextElement(text);
        html.add(body);
        body.add(p);
        p.add(textElement);
        return html.toString();
    }
}
```

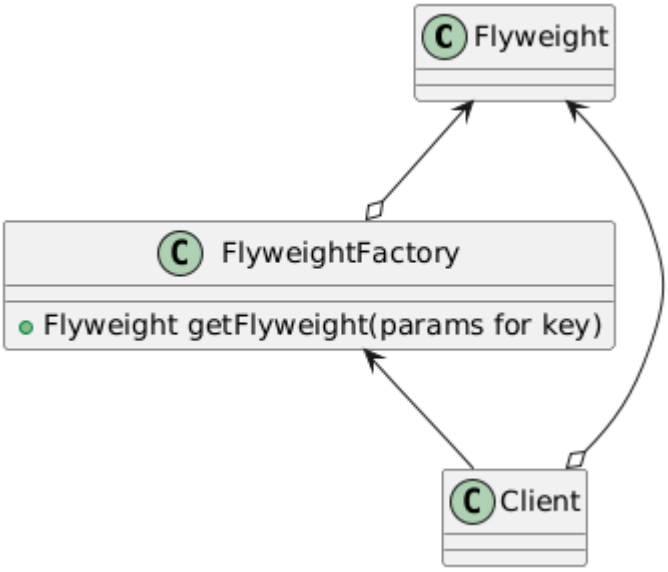
```
class MarkupTest {

    @Test
    void test_markup() {
        Markup markup = new Markup();
        String markupText= markup.createMarkup("Hello World!");

        String expectedText = "<html>"
                              + "<body><p>Hello World!</p></body>"
                              + "</html>";

        assertEquals(expectedText, markupText);
    }
}
```

Flyweight Pattern

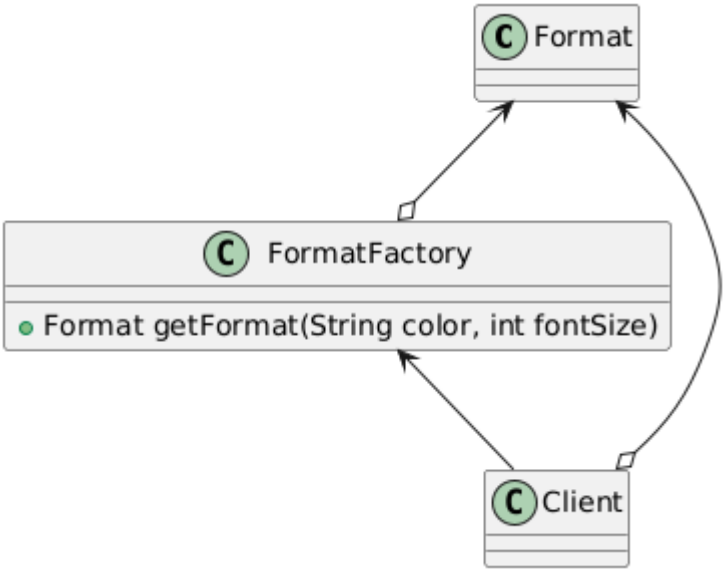
GoF Definition

Use sharing to support large numbers of fine-grained objects efficiently.

Basic Components



UML Representation




```
public class Format {
    private int fontSize;
    private String color;

    public Format(String color, int fontSize) {
        this.color = color;
```

```
        this.fontSize = fontSize;
    }

    public int getFontSize() {
        return fontSize;
    }

    public String getColor() {
        return color;
    }
}
```

```
public class FormatFactory {
    private Map<String, Format> formatMap = new HashMap<>();

    private String getKey(String color, int fontSize) {
        return color + fontSize;
    }

    public int getCount() {
        return formatMap.size();
    }

    public Format getFormat(String color, int fontSize) {
        String key = getKey(color, fontSize);
        Format format = formatMap.get(key);
        if( format == null ) {
            format = new Format(color, fontSize);
            formatMap.put(key, format);
        }
        return format;
    }
}
```

```
class FormatFactoryTest {

    @Test
    void test_required_objects_created_only() {
        List<Format> formatList = new ArrayList<>();
        FormatFactory formatFactory = new FormatFactory();

        formatList.add(formatFactory.getFormat("Blue", 20)); // 1
        formatList.add(formatFactory.getFormat("Blue", 20));
        formatList.add(formatFactory.getFormat("Blue", 20));
        formatList.add(formatFactory.getFormat("White", 10)); // 2
        formatList.add(formatFactory.getFormat("White", 10));
        formatList.add(formatFactory.getFormat("White", 30)); // 3
        formatList.add(formatFactory.getFormat("Red", 24)); // 4
```

```
        formatList.add(formatFactory.getFormat("Red", 24));
        formatList.add(formatFactory.getFormat("Red", 24));
        formatList.add(formatFactory.getFormat("Red", 32)); // 5

        assertEquals(5, formatFactory.getCount());
    }

}
```

References in JDK

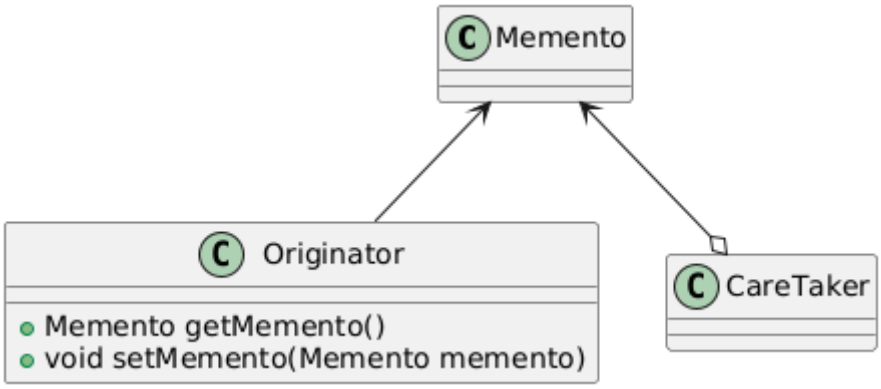
- java.lang.Integer#valueOf(int) (also Boolean, Byte, Character, Short, Long and BigDecimal)

Memento Pattern

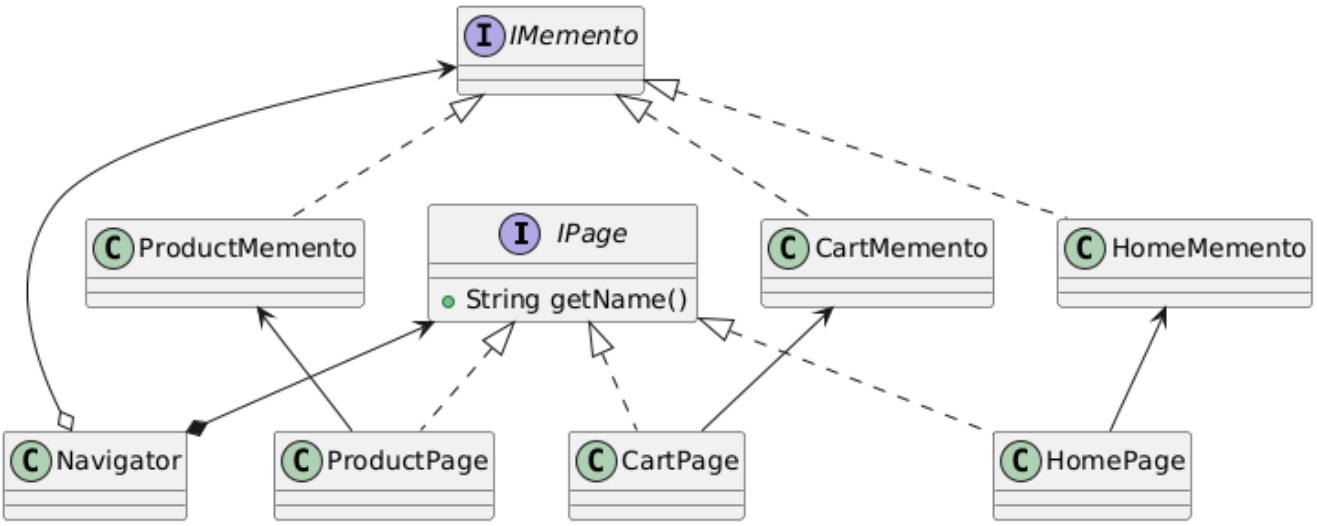
GoF Definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

Basic Components



UML Representation

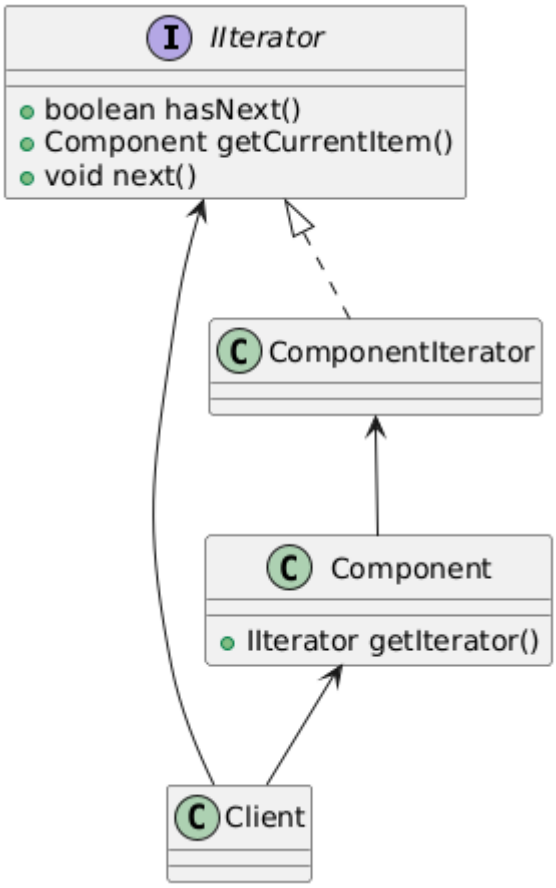


Iterator Pattern

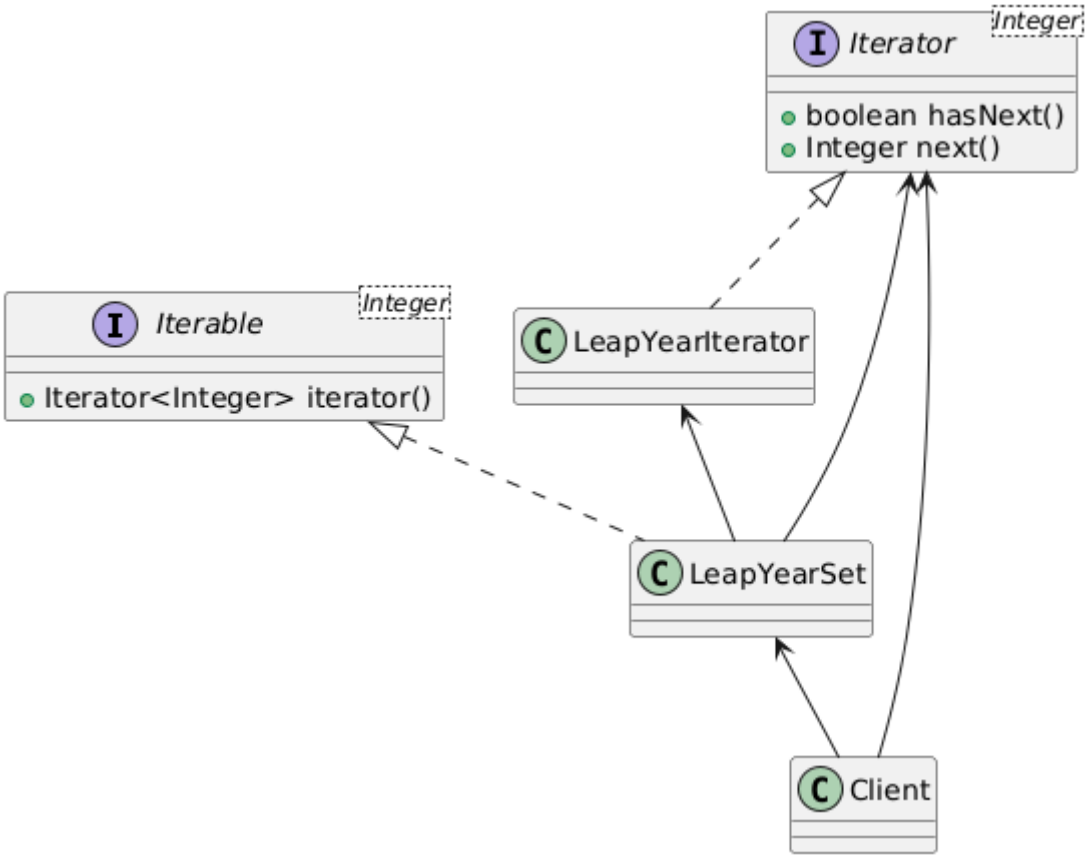
GoF Definition

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation. (Also known as Cursor)

Basic Components



UML Representation

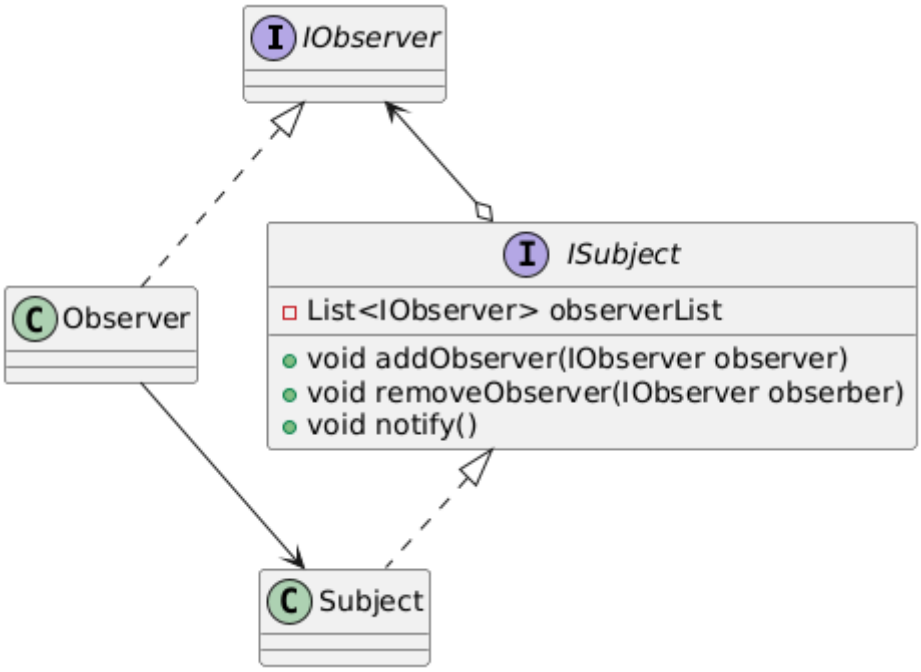

Iterator
Integer
boolean hasNext()
Integer next()
Iterable
Integer
Iterator<Integer> iterator()
LeapYearIterator
LeapYearSet
Client

Observer Pattern

GoF Definition

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Basic Components

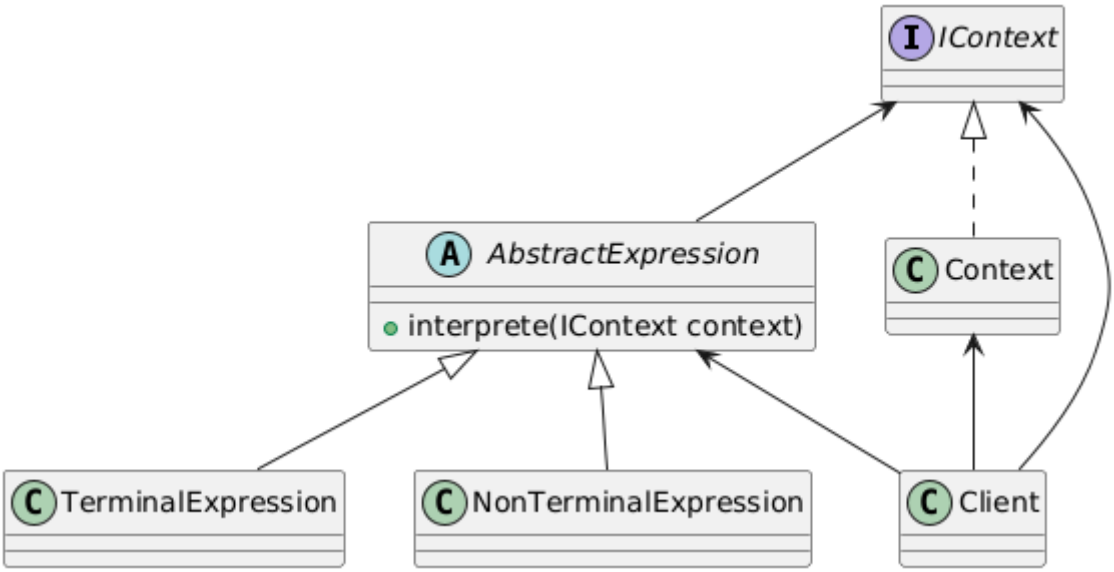


Interpreter Pattern

GoF Definition

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Basic Components



Custom DSL and Evaluation

For this example we consider a custom DSL for creating expressions which could be evaluated. The DSL will have elements separated by space, which will help us tokenize the expression quickly.

Operators we will recognize

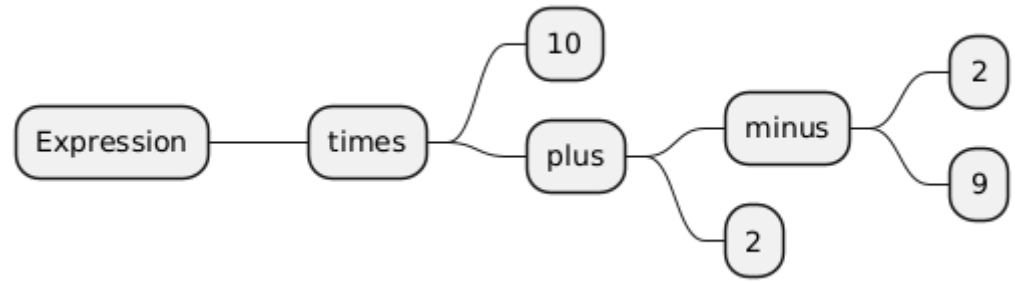
- plus - for addition
- minus - for subtraction
- times - for multiplication
- by - for division
- (- for open parenthesis
-) - for closed parenthesis

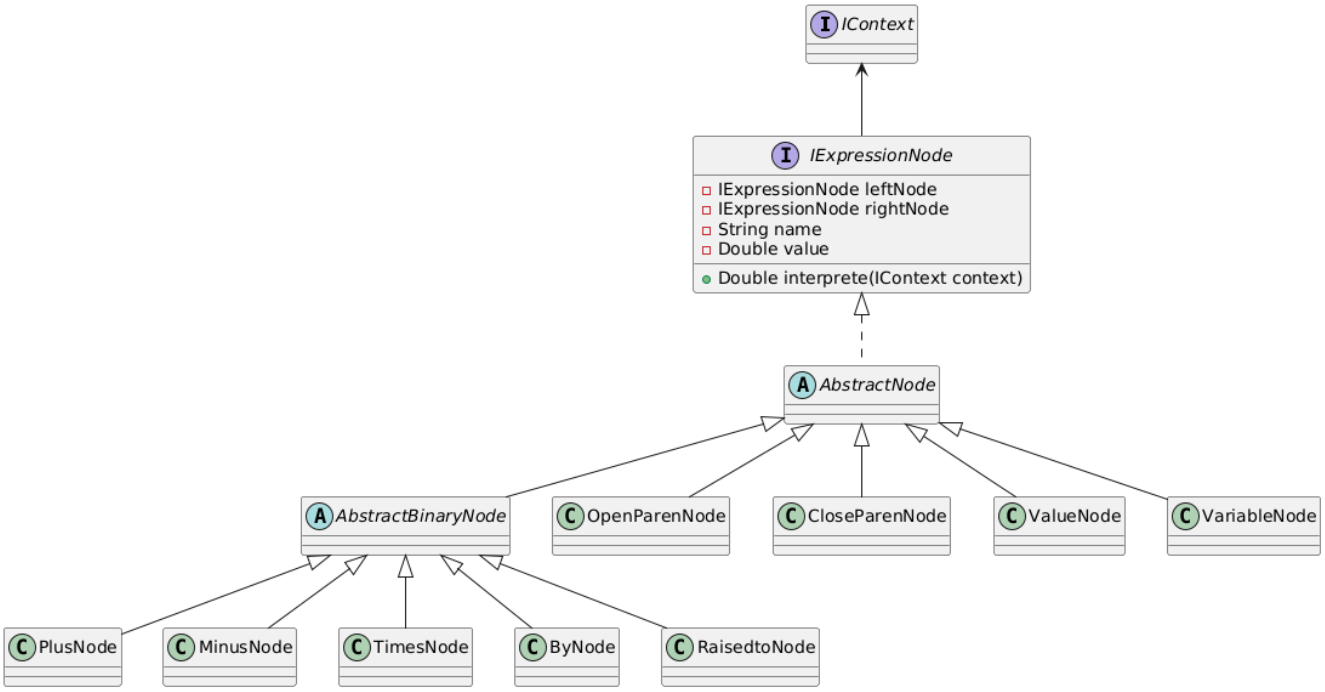
All other characters will be considered as either variables or numbers. If the character sequence could be parsed into a number it will be assumed as number. If the character sequence is not parseable as number it will be considered as a variable. The value of variable will be fetched from the context later while interpreting time.

Example Expression

2 plus (9 minus 2) times 10

This expression shall be converted into Abstract Syntax Tree (AST) so the root of the tree can be called. The root of the tree when called with interprete method will delegate the inpretation to all child elements.



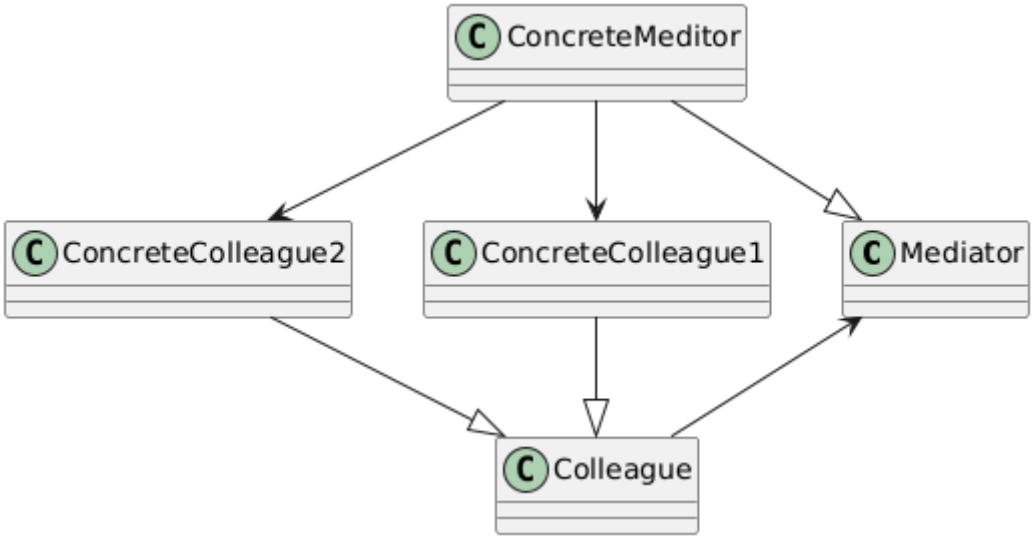


Mediator Pattern

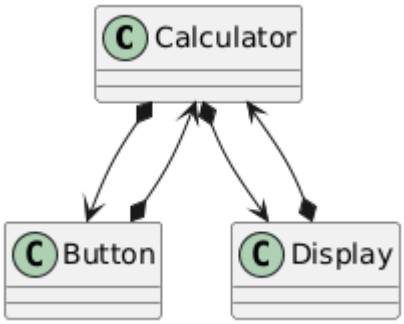
GoF Definition

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

Basic Components



UML representation




```
public class Display {
    private String text;
    private Calculator calculator;

    public Display(Calculator calculator) {
        this.calculator = calculator;
    }

    public String getText() {
        return text;
    }
```

```
    public void setText(String text) {
        this.text = text;
    }

    public void click() {
        calculator.onClick("display");
    }
}
```

```
public class Button {
    private String title;
    private Calculator calculator;

    public Button(Calculator calculator, String title) {
        this.calculator = calculator;
        this.title = title;
    }

    public void click() {
        calculator.onClick(title);
    }
}
```

```
public class Calculator {
    private Display display;
    private Button one;
    private Button zero;
    private Button plus;
    private Button minus;
    private Button equalTo;
    private String previousNumberText = "";
    private String currentNumberText = "";
    private String operator = null;

    public Calculator() {
        display = new Display(this);
        one = new Button(this, "1");
        zero = new Button(this, "0");
        plus = new Button(this, "plus");
        minus = new Button(this, "minus");
        equalTo = new Button(this, "=");
    }

    public Button getOne() {
        return one;
    }

    public Button getZero() {
```

```
        return zero;
    }

    public Button getPlus() {
        return plus;
    }

    public Button getMinus() {
        return minus;
    }

    public Button getEqualTo() {
        return equalTo;
    }

    public Display getDisplay() {
        return display;
    }

    public void onClick(String title) {
        switch(title) {
        case "1":
        case "0":
            storeDigit(title);
            break;
        case "plus":
        case "minus":
            storeOperator(title);
            break;
        case "=":
            showResult();
            break;
        case "display":
            reset();
            break;
        default:
            break;
        }
    }

    private void storeDigit(String title) {
        if( operator == null ) {
            previousNumberText += title;
            display.setText(previousNumberText);
        } else {
            currentNumberText += title;
            display.setText(currentNumberText);
        }
    }

    private void storeOperator(String title) {
```

```
        if(operator != null ) {
            showResult();
        }
        operator = title;
    }

    private void reset() {
        previousNumberText = "";
        currentNumberText = "";
        operator = null;
        display.setText("");
    }

    private void showResult() {
        if(!isValidStateForResult())
            return;

        Integer num1 = Integer.parseInt(previousNumberText, 2);
        Integer num2 = Integer.parseInt(currentNumberText, 2);
        Integer result;
        if( operator.equals("plus")) {
            result = num1 + num2;
        } else {
            result = num1 + num2;
        }
        previousNumberText = Integer.toBinaryString(result);
        display.setText(previousNumberText);
        currentNumberText = "";
        operator = null;
    }

    private boolean isValidStateForResult() {
        if(previousNumberText == null || previousNumberText.length() == 0) {
            return false;
        }
        if(currentNumberText == null || currentNumberText.length() == 0) {
            return false;
        }
        if( operator == null) {
            return false;
        }

        return true;
    }
}
```

```
class CalculatorTest {
    @Test
    void test_calculator() {
        Calculator calculator = new Calculator();
```

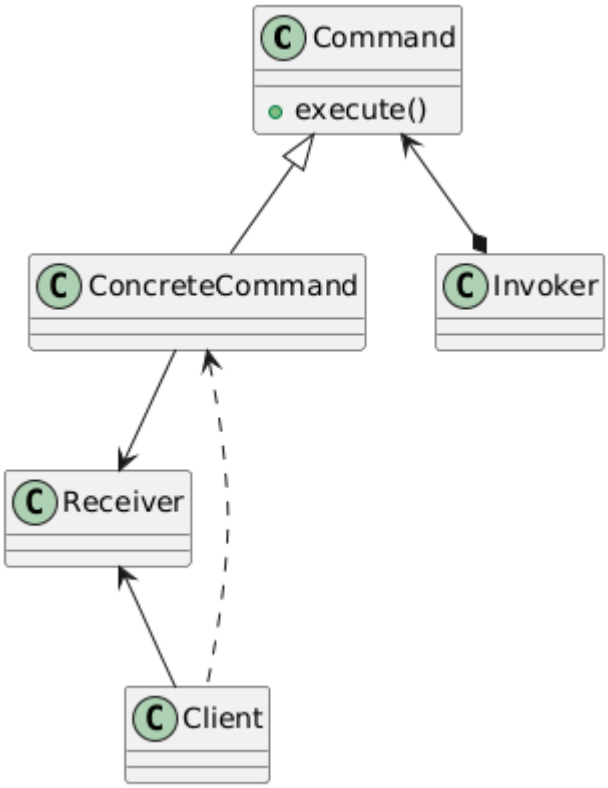
```
        calculator.getZero().click();       // |-----0-|
        assertEquals("0", calculator.getDisplay().getText());

        calculator.getOne().click();        // |----01-|
        assertEquals("01", calculator.getDisplay().getText());

        calculator.getPlus().click();       // |----01-|
        assertEquals("01", calculator.getDisplay().getText());

        calculator.getOne().click();        // |-----1-|
        assertEquals("1", calculator.getDisplay().getText());

        calculator.getZero().click();       // |----10-|
        assertEquals("10", calculator.getDisplay().getText());

        calculator.getEqualTo().click();    // |----11-|
        assertEquals("11", calculator.getDisplay().getText());
    }
}
```

Command Pattern

GoF Definition

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Basic Components




```
public class Command {
    private String title;
    private Calculator calculator;

    public Command(Calculator calculator, String title) {
        this.calculator = calculator;
        this.title = title;
    }

    public void execute() {
        calculator.onClick(title);
    }
}
```

```
public class Button {
    private String title;
    private Calculator calculator;
```

```
    public Button(Calculator calculator, String title) {
        this.calculator = calculator;
        this.title = title;
    }

    public void click() {
        calculator.onCommand(title);
    }
}
```

```
public class Display {
    private String text;
    private Calculator calculator;

    public Display(Calculator calculator) {
        this.calculator = calculator;
    }

    public String getText() {
        return text;
    }

    public void setText(String text) {
        this.text = text;
    }

    public void click() {
        calculator.onCommand("display");
    }
}
```

```
public class Calculator {
    private Display display;
    private Button one;
    private Button zero;
    private Button plus;
    private Button minus;
    private Button equalTo;
    private Button undo;
    private Button redo;
    private String previousNumberText = "";
    private String currentNumberText = "";
    private String operator = null;
    private Stack<Command> undoStack = new Stack<>();
    private Stack<Command> redoStack = new Stack<>();

    public Calculator() {
```

```
        display = new Display(this);
        one = new Button(this, "1");
        zero = new Button(this, "0");
        plus = new Button(this, "plus");
        minus = new Button(this, "minus");
        equalTo = new Button(this, "=");
        undo = new Button(this, "undo");
        redo = new Button(this, "redo");
    }

    public Button getOne() {
        return one;
    }

    public Button getZero() {
        return zero;
    }

    public Button getPlus() {
        return plus;
    }

    public Button getMinus() {
        return minus;
    }

    public Button getEqualTo() {
        return equalTo;
    }

    public Button getUndo() {
        return undo;
    }

    public Button getRedo() {
        return redo;
    }

    public Display getDisplay() {
        return display;
    }

    public void onCommand(String title) {
        switch(title) {
            case "undo":
                undo();
                break;
            case "redo":
                redo();
                break;
            default:
```

```
                addCommand(title);
                onClick(title);
                break;
        }
    }

    public void onClick(String title) {
        switch(title) {
        case "1":
        case "0":
            storeDigit(title);
            break;
        case "plus":
        case "minus":
            storeOperator(title);
            break;
        case "=":
            showResult();
            break;
        case "display":
            reset();
            break;
        default:
            break;
        }
    }

    private void addCommand(String title) {
        undoStack.push(new Command(this, title));
        redoStack.clear();
    }

    private void undo() {
        if( !undoStack.empty()) {
            redoStack.push(undoStack.pop());
            replay();
        }
    }

    private void redo() {
        if( !redoStack.empty()) {
            Command command = redoStack.pop();
            undoStack.push(command);
            command.execute();
        }
    }

    private void replay() {
        reset();
        if( !undoStack.empty()) {
            for(Command command : undoStack) {
```

```
                command.execute();
            }
        }
    }

    private void storeDigit(String title) {
        if( operator == null ) {
            previousNumberText += title;
            display.setText(previousNumberText);
        } else {
            currentNumberText += title;
            display.setText(currentNumberText);
        }
    }

    private void storeOperator(String title) {
        if(operator != null ) {
            showResult();
        }
        operator = title;
    }

    private void reset() {
        previousNumberText = "";
        currentNumberText = "";
        operator = null;
        display.setText("");
    }

    private void showResult() {
        if(!isValidStateForResult())
            return;

        Integer num1 = Integer.parseInt(previousNumberText, 2);
        Integer num2 = Integer.parseInt(currentNumberText, 2);
        Integer result;
        if( operator.equals("plus")) {
            result = num1 + num2;
        } else {
            result = num1 + num2;
        }
        previousNumberText = Integer.toBinaryString(result);
        display.setText(previousNumberText);
        currentNumberText = "";
        operator = null;
    }

    private boolean isValidStateForResult() {
        if(previousNumberText == null || previousNumberText.length() == 0) {
            return false;
        }
```

```
        if(currentNumberText == null || currentNumberText.length() == 0) {
            return false;
        }
        if( operator == null) {
            return false;
        }

        return true;
    }
}
```

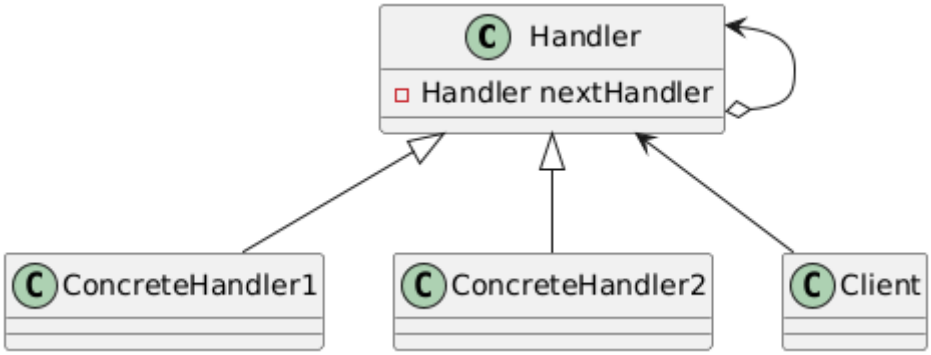
```
class CalculatorTest {
    @Test
    void test_calculator() {
        Calculator calculator = new Calculator();

        calculator.getOne().click();         // |-----1-|
        assertEquals("1", calculator.getDisplay().getText());

        calculator.getZero().click();        // |----10-|
        assertEquals("10", calculator.getDisplay().getText());

        calculator.getUndo().click();        // |-----1-|
        assertEquals("1", calculator.getDisplay().getText());

        calculator.getRedo().click();        // |----10-|
        assertEquals("10", calculator.getDisplay().getText());
    }
}
```

Chain of Responsibility Pattern

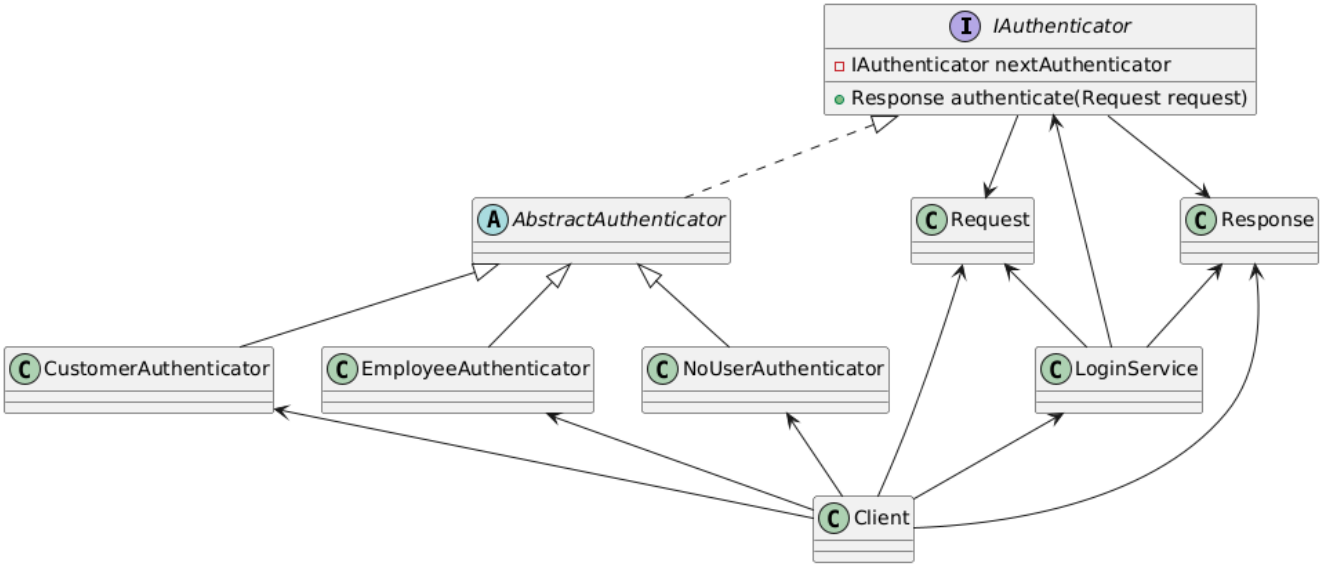
GoF Definition

Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

Basic Components



UML Representation




```
public class Request {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
```

```
    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

```
public class Response {
    private String result;
    private String message;

    public String getResult() {
        return result;
    }

    public void setResult(String result) {
        this.result = result;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

```
public interface IAuthenticator {
    IAuthenticator setNext(IAuthenticator authenticator);
    Response authenticate(Request request);
}
```

```
public abstract class AbstractAuthenticator implements IAuthenticator {
    private IAuthenticator nextAuthenticator;

    public IAuthenticator setNext(IAuthenticator authenticator) {
        nextAuthenticator = authenticator;
        return authenticator;
    }

    public Response authenticate(Request request) {
        if( nextAuthenticator != null ) {
            return nextAuthenticator.authenticate(request);
```

```
        }

        return null;
    }
}
```

```
public class CustomerAuthenticator extends AbstractAuthenticator {

    private String[] customerList = {
        "customer1@gmail.com",
        "customer2@rediffmain.com"
    };

    public Response authenticate(Request request) {
        for(String customer : customerList) {
            if(customer.equals(request.getUsername())) {
                Response response = new Response();
                response.setResult("SUCCESS");
                response.setMessage("Customer Login Successful.");
                return response;
            }
        }

        return super.authenticate(request);
    }
}
```

```
public class EmployeeAuthenticator extends AbstractAuthenticator {

    private String[] employeeList = {
        "employee1@company.com",
        "employee2@company.com"
    };

    public Response authenticate(Request request) {
        for(String enployee : employeeList) {
            if(enployee.equals(request.getUsername())) {
                Response response = new Response();
                response.setResult("SUCCESS");
                response.setMessage("Employee Login Successful.");
                return response;
            }
        }

        return super.authenticate(request);
    }
}
```

```
public class NoUserAuthenticator extends AbstractAuthenticator {
    public Response authenticate(Request request) {
        Response response = new Response();
        response.setResult("FAILURE");
        response.setMessage("No User Found.");
        return response;
    }
}
```

```
public class LoginService {
    private IAuthenticator authenticator;

    public LoginService(IAuthenticator authenticator) {
        this.authenticator = authenticator;
    }
    public Response login(String username, String password) {
        Request request = new Request();
        request.setUsername(username);
        request.setPassword(password);
        return authenticator.authenticate(request);
    }
}
```

```
class LoginServiceTest {

    private LoginService target;

    @BeforeEach
    public void beforeEach() {
        IAuthenticator authenticator = new EmployeeAuthenticator();
        authenticator
            .setNext(new CustomerAuthenticator())
            .setNext(new NoUserAuthenticator());
        target = new LoginService(authenticator);
    }

    @Test
    void test_login_customer() {
        Response response = target.login("customer1@gmail.com", "password");
        assertEquals("SUCCESS", response.getResult());
        assertEquals("Customer Login Successful.", response.getMessage());
    }

    @Test
    void test_login_employee() {
        Response response = target.login("employee1@company.com", "password");
        assertEquals("SUCCESS", response.getResult());
        assertEquals("Employee Login Successful.", response.getMessage());
```

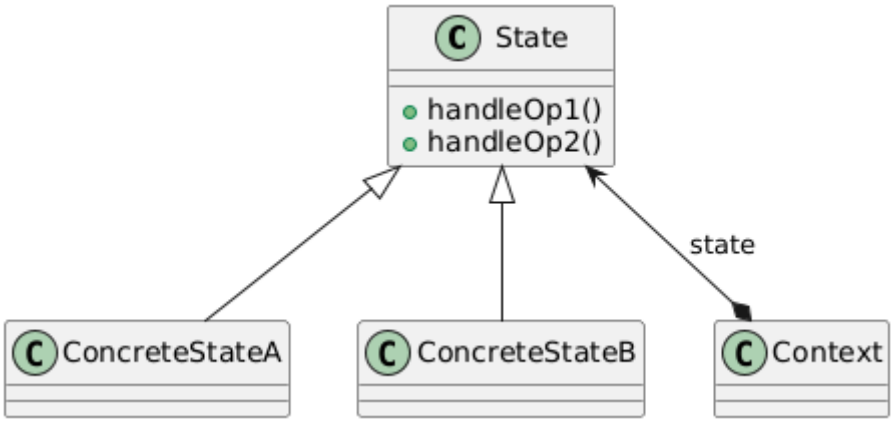
```
    }

    @Test
    void test_login_unknown() {
        Response response = target.login("somebody@company.com", "password");
        assertEquals("FAILURE", response.getResult());
        assertEquals("No User Found.", response.getMessage());
    }
}
```

State Pattern

GoF Definition

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Basic Components

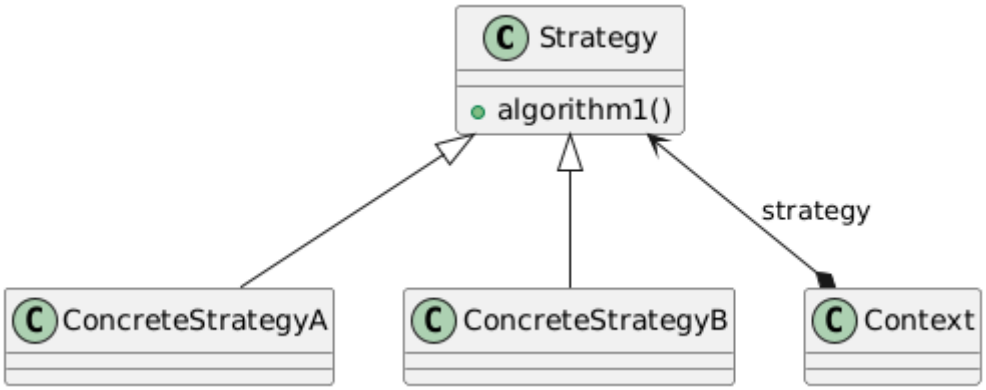


Strategy Pattern

GoF Definition

Define a family of algorithms, encapsulate each one, and make them interchange- able. Strategy lets the algorithm vary independently from clients that use it.

Basic Components

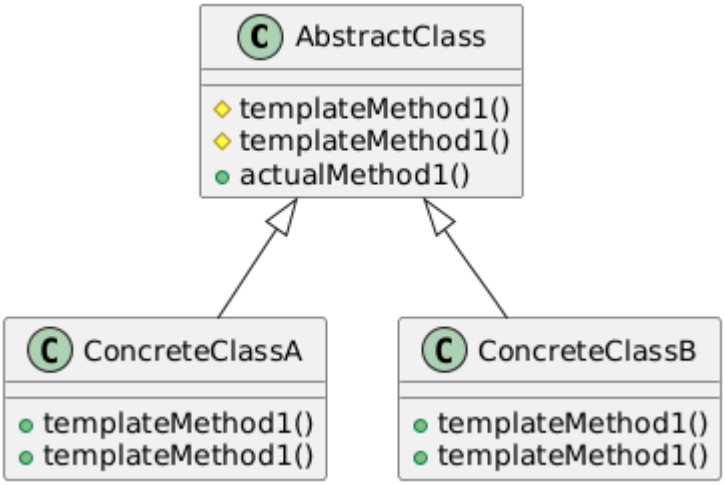


Template Method Pattern

GoF Definition

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Basic Components

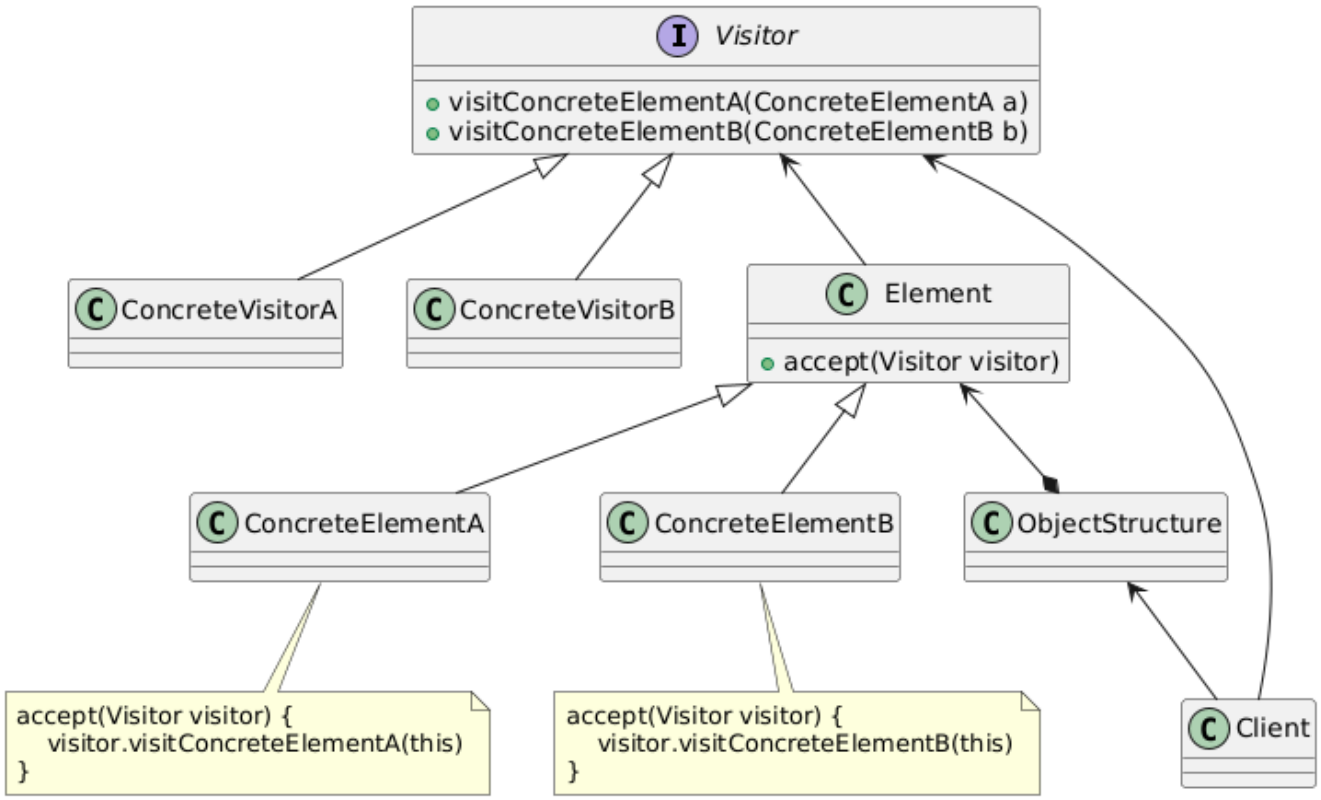


Visitor Pattern

GoF Definition

Represent an operation to be performed on the elements of an object structure.Vis- itor lets you define a new operation without changing the classes of the elements on which it operates.

Basic Components




```
public abstract class AbstractVisitor {
    public void visitHTMLElement(HTMLElement element) {
        // empty method
    }

    public void visitBodyElement(BodyElement element) {
        // empty method
    }

    public void visitPElement(PElement element) {
        // empty method
    }

    public void visitTextElement(TextElement element) {
        // empty method
    }
```

```
}
```

```
public class HTMLElement {
    protected String tagName;
    private List<HTMLElement> children = new ArrayList<>();

    public HTMLElement() {
        this("html");
    }

    public void visitChildren(AbstractVisitor visitor) {
        for(HTMLElement element : children) {
            element.accept(visitor);
        }
    }

    public void accept(AbstractVisitor visitor) {
        visitChildren(visitor);
        visitor.visitHTMLElement(this);
    }

    public HTMLElement(String tagName) {
        this.tagName = tagName;
    }

    public void add(HTMLElement element) {
        children.add(element);
    }

    public String toString() {
        StringBuilder builder = new StringBuilder();
        builder.append("<");
        builder.append(tagName);
        builder.append(">");
        for(HTMLElement element : children) {
            builder.append(element.toString());
        }
        builder.append("</");
        builder.append(tagName);
        builder.append(">");

        return builder.toString();
    }
}
```

```
public class BodyElement extends HTMLElement {
    public BodyElement() {
        super("body");
```

```
    }

    public void accept(AbstractVisitor visitor) {
        visitChildren(visitor);
        visitor.visitBodyElement(this);
    }
}
```

```
public class PElement extends HTMLElement {
    public PElement() {
        super("p");
    }

    public void accept(AbstractVisitor visitor) {
        visitChildren(visitor);
        visitor.visitPElement(this);
    }
}
```

```
public class TextElement extends HTMLElement {
    public TextElement(String text) {
        super(text);
    }

    public String getText() {
        return tagName;
    }

    public void setText(String text) {
        tagName = text;
    }

    public String toString() {
        return tagName;
    }

    public void accept(AbstractVisitor visitor) {
        visitor.visitTextElement(this);
    }
}
```

```
public class Markup {
    public HTMLElement createMarkup(String text) {
        HTMLElement html = new HTMLElement();
        BodyElement body = new BodyElement();
        PElement p = new PElement();
        TextElement textElement = new TextElement(text);
```

```
        html.add(body);
        body.add(p);
        p.add(textElement);
        return html;
    }
}
```

```
public class BasicVisitor extends AbstractVisitor {
    private List<String> visitedList = new ArrayList<>();

    public List<String> getList() {
        return visitedList;
    }

    public void visitHTMLElement(HTMLElement element) {
        visitedList.add("Visited HTML: " + element.toString());
        visitedList.add("\n--\n");
    }

    public void visitBodyElement(BodyElement element) {
        visitedList.add("Visited Body: " + element.toString());
        visitedList.add("\n--\n");
    }

    public void visitPElement(PElement element) {
        visitedList.add("Visited P: " + element.toString());
        visitedList.add("\n--\n");
    }

    public void visitTextElement(TextElement element) {
        visitedList.add("Visited Text: " + element.toString());
        visitedList.add("\n--\n");
    }

}
```

```
class VisitorTest {

    @Test
    void test_markup_visitor() {
        Markup markup = new Markup();
        HTMLElement rootElement = markup.createMarkup("Hello World!");

        BasicVisitor visitor = new BasicVisitor();
        rootElement.accept(visitor);

        String[] expectedValues = {
            "Visited Text: Hello World!",
```

```
            "\n--\n",
            "Visited P: <p>Hello World!</p>",
            "\n--\n",
            "Visited Body: <body><p>Hello World!</p></body>",
            "\n--\n",
            "Visited HTML: <html><body><p>Hello World!</p></body></html>",
            "\n--\n"
        };

        for(int index = 0; index < visitor.getList().size(); index++) {
            assertEquals(expectedValues[index], visitor.getList().get(index));
        }
    }

    @Test
    void test_markup_textenhancervisitor() {
        Markup markup = new Markup();
        HTMLElement rootElement = markup.createMarkup("Hello World!");

        BasicVisitor visitor = new BasicVisitor();
        TextEnhancerVisitor textEnhancerVisitor = new TextEnhancerVisitor();
        rootElement.accept(textEnhancerVisitor);
        rootElement.accept(visitor);

        String[] expectedValues = {
            "Visited Text: Hello World!(New)",
            "\n--\n",
            "Visited P: <p>Hello World!(New)</p>",
            "\n--\n",
            "Visited Body: <body><p>Hello World!(New)</p></body>",
            "\n--\n",
            "Visited HTML: <html><body><p>Hello World!(New)</p></body></html>",
            "\n--\n"
        };

        for(int index = 0; index < visitor.getList().size(); index++) {
            assertEquals(expectedValues[index], visitor.getList().get(index));
        }
    }
}
```

Example Conventions

- CSharpHandbook on Github by Marco van Ballmoose
- Microsoft Framework Design Guidelines

References

- Abran, Alain, editor. Guide to the Software Engineering Body of Knowledge, 2004 Version: SWEBOK; a Project of the IEEE Computer Society Professional Practices Committee. IEEE Computer Society, 2004.
- Freeman, Eric, et al., editors. Head First Design Patterns. O'Reilly, 2004.
- Gamma, Erich, editor. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Sarcar, Vaskaran. Java Design Patterns: A Tour of 23 Gang of Four Design Patterns in Java. Apress, 2016.
- Timms, Simon. Mastering Javascript Design Patterns. Packt Publishing, 2014.
- Martin, Robert C., editor. Clean Code: A Handbook of Agile Software Craftsmanship. Prentice Hall, 2009.
- Hall, Gary McLean. Adaptive Code: Agile Coding with Design Patterns and SOLID Principles. Second edition, Microsoft Press, 2017.
- Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.
- Leitner, Helmut. Pattern Theory: Introduction and Perspectives on the Tracks of Christopher Alexander. First printing, Helmut Leitner, HLS Software, 2015.
- Thomas, David, and Andrew Hunt. The Pragmatic Programmer, 20th Anniversary Edition: Journey to Mastery. Second edition, Addison-Wesley, 2019.
- Kerievsky, Joshua. Refactoring to Patterns. 5 print, Addison-Wesley, 2007.
- Thomas, David, and Andrew Hunt. The Pragmatic Programmer, 20th Anniversary Edition: Journey to Mastery. Second edition, Addison-Wesley, 2019.
- Software Architecture Guide by Free Technology Academy
- Martin Fowler's Catalog of Patterns of Enterprise Application Architecture
- Coding Geek Design Pattern category posts
- Refactoring Guru Design Patterns Catalog
- Martin Fowlers Article on Patterns
- A Pattern Language for Pattern Writing by Gerard Meszaros, Jim Doble
- Design Pattern tag articles on StackAbuse.com site Structural Behavioral Catalog
- Examples from JDK community post on StackOverflow.com
- Popular mnemonic for Design Patterns from Ajit Mungale from 2003
- About Interpreter Pattern and its difficult nature.
- Design Patterns Cheatsheet by Jason S. McDonald
- Design pattern examples at OODesign.com
- Allen Holub Design Patterns Reference PDF

Appendix A: Mnemonic to remember design patterns

Some details here

Popular Mnemonic from Ajit Mungale's article

Interesting mnemonic for the design patterns is available on internet from a post in 2003. I was able to find reference to thisarticle from 2003 and then it is copied and published in many articles. The real origin of this mnemonic is unknown.

Creational Patterns

- Abstract Factory: Creates an instance of several families of classes
- Builder: Separates object construction from its representation
- Factory Method: Creates an instance of several derived classes
- Prototype: A fully initialized instance to be copied or cloned
- Singleton: A class in which only a single instance can exist

Mnemonic: ABFPS (Abraham Became First President of States).

Structural Patterns

- Adapter: Match interfaces of different classes
- Bridge: Separates an object's abstraction from its implementation
- Composite: A tree structure of simple and composite objects
- Decorator: Add responsibilities to objects dynamically
- Façade: A single class that represents an entire subsystem
- Flyweight: A fine-grained instance used for efficient sharing
- Proxy: An object representing another object

Mnemonic: ABCDFFP

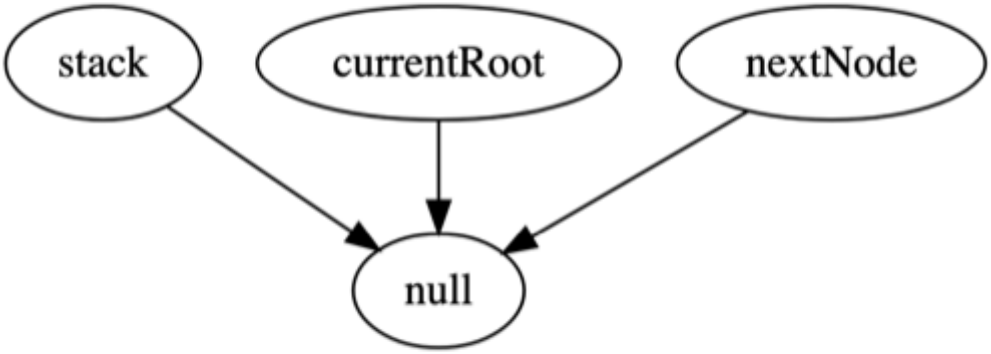
Behavioral Patterns

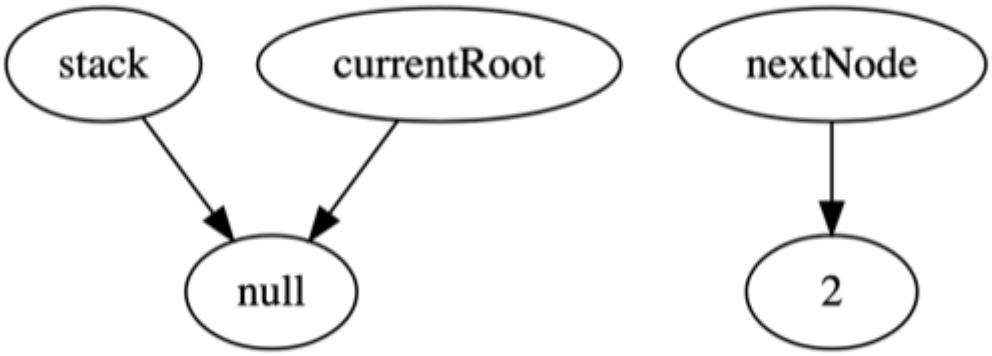
- Mediator: Defines simplified communication between classes
- Memento: Capture and restore an object's internal state
- Interpreter: A way to include language elements in a program
- Iterator: Sequentially access the elements of a collection
- Chain of Resp: A way of passing a request between a chain of objects
- Command: Encapsulate a command request as an object

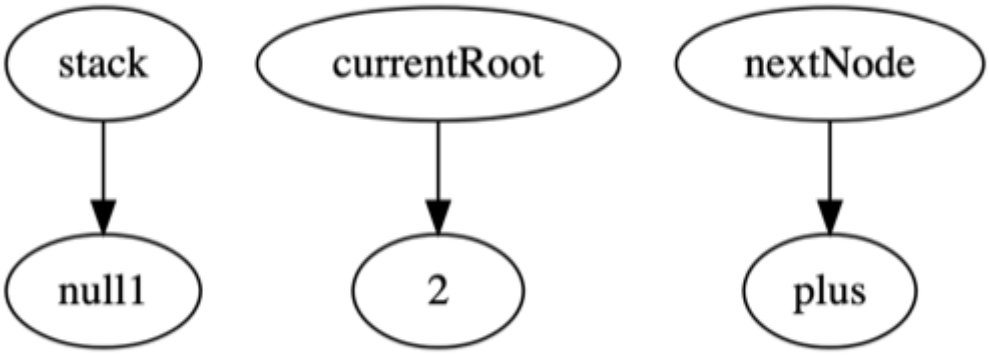
- State: Alter an object's behavior when its state changes
- Strategy: Encapsulates an algorithm inside a class
- Observer: A way of notifying change to a number of classes
- Template Method: Defer the exact steps of an algorithm to a subclass
- Visitor: Defines a new operation to a class without change

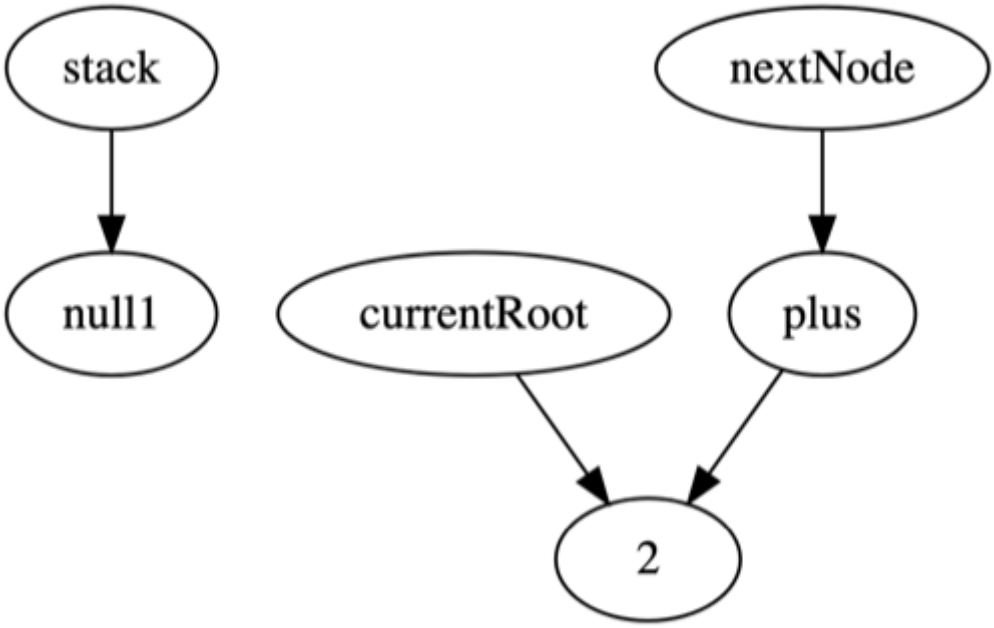
Mnemonic: 2 MICS On TV (MMIICCSSOTV)

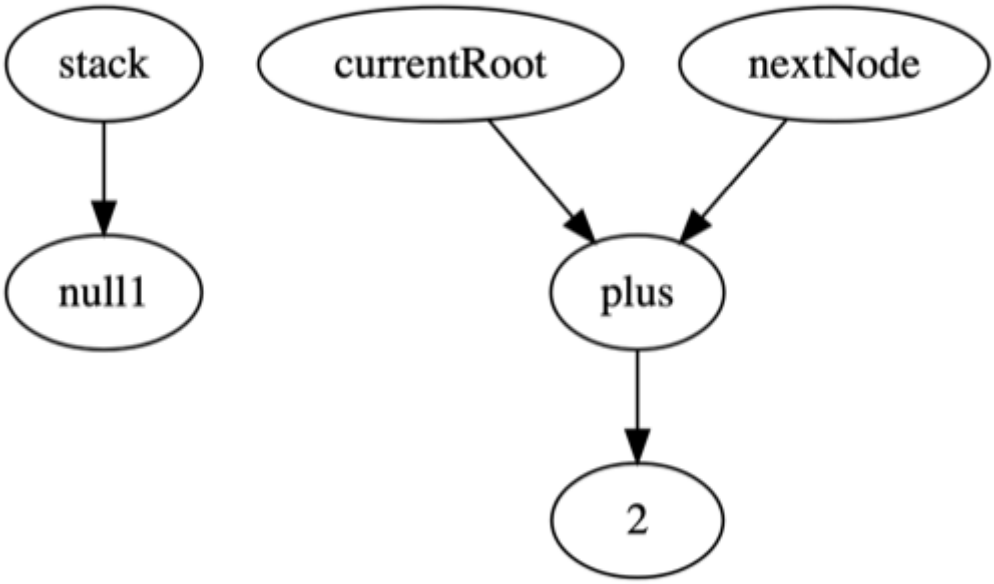
Appendix B: Building Abstract Syntax Tree

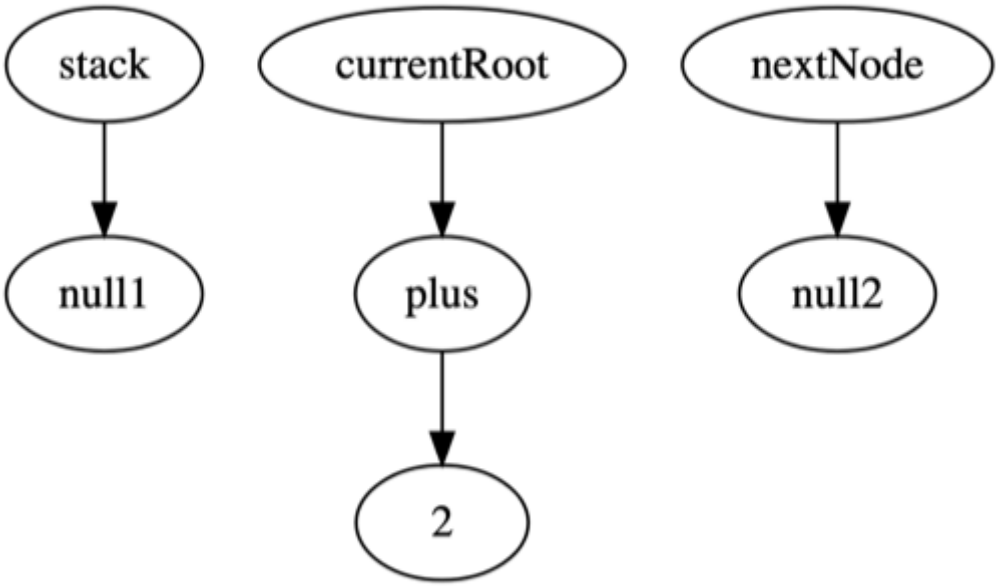


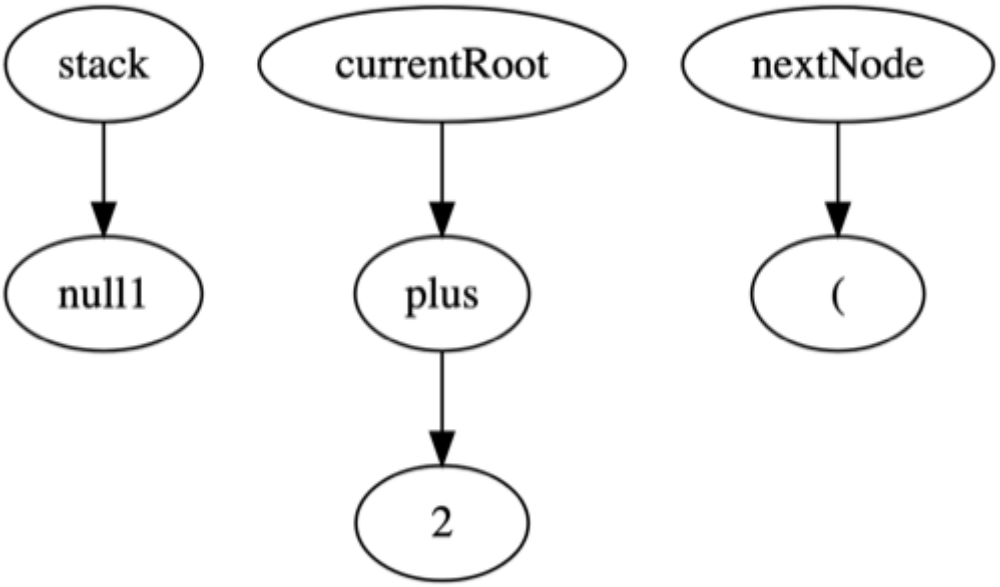

stack
currentRoot
nextNode
null
2

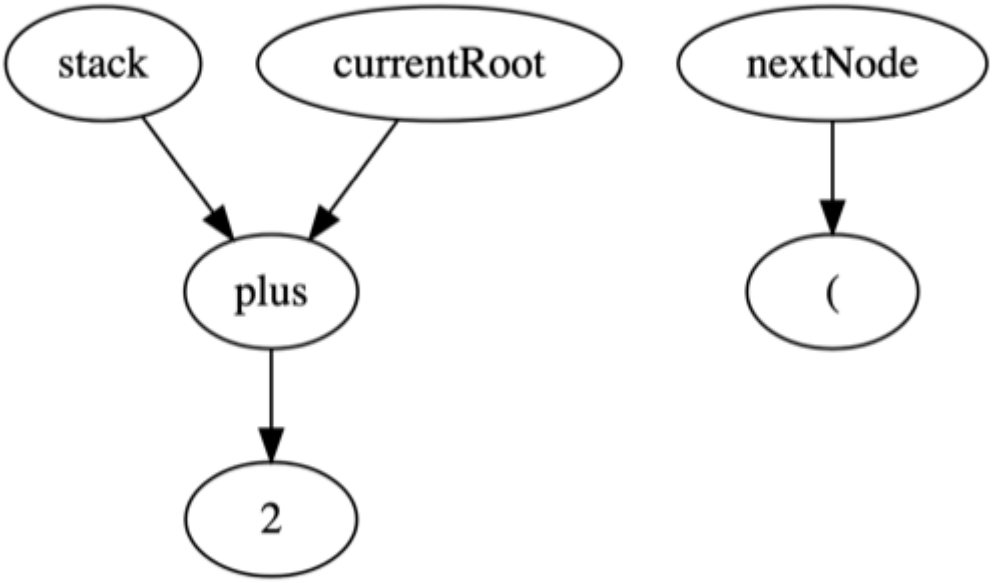

stack
currentRoot
nextNode
null1
2
plus

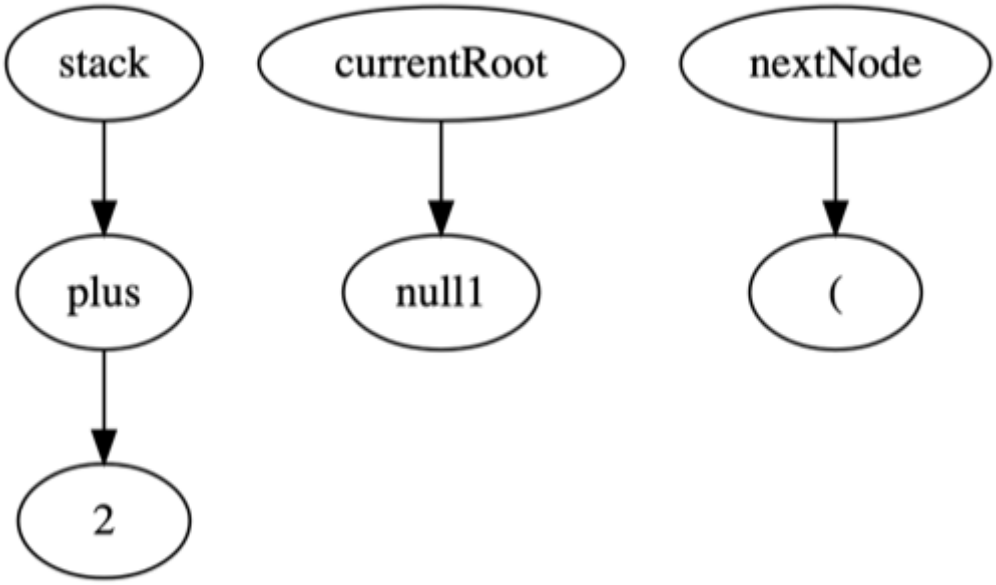

stack
nextNode
null1
currentRoot
plus
2

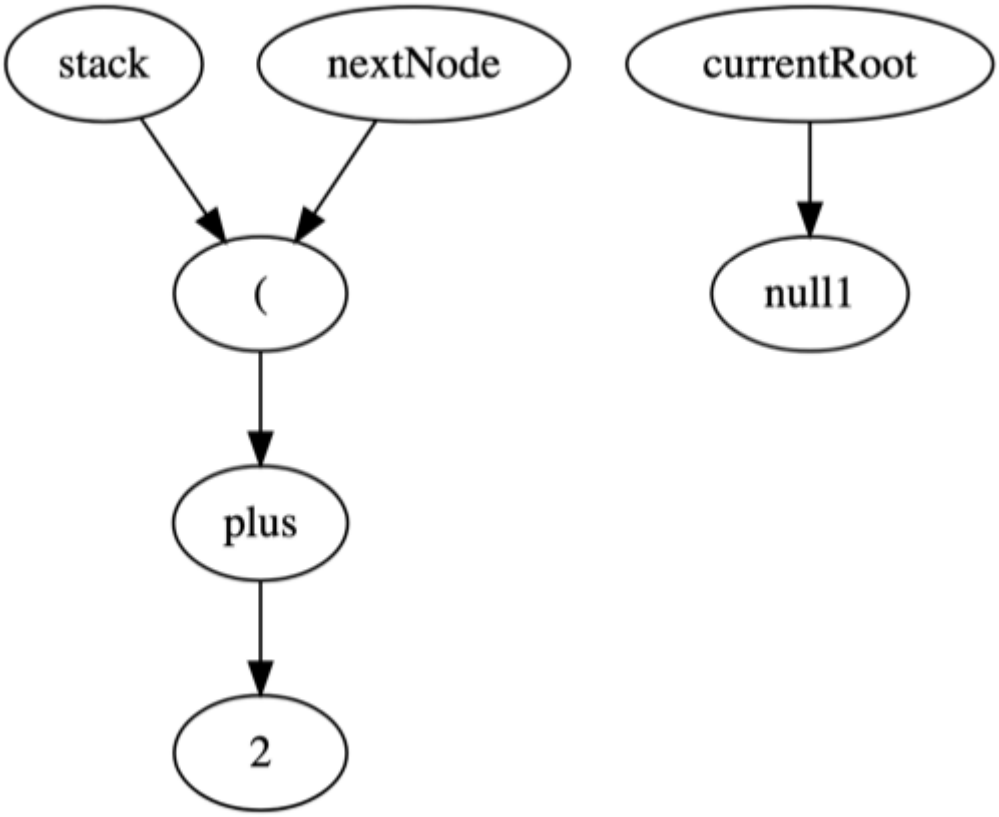

stack
currentRoot
nextNode
null1
plus
2

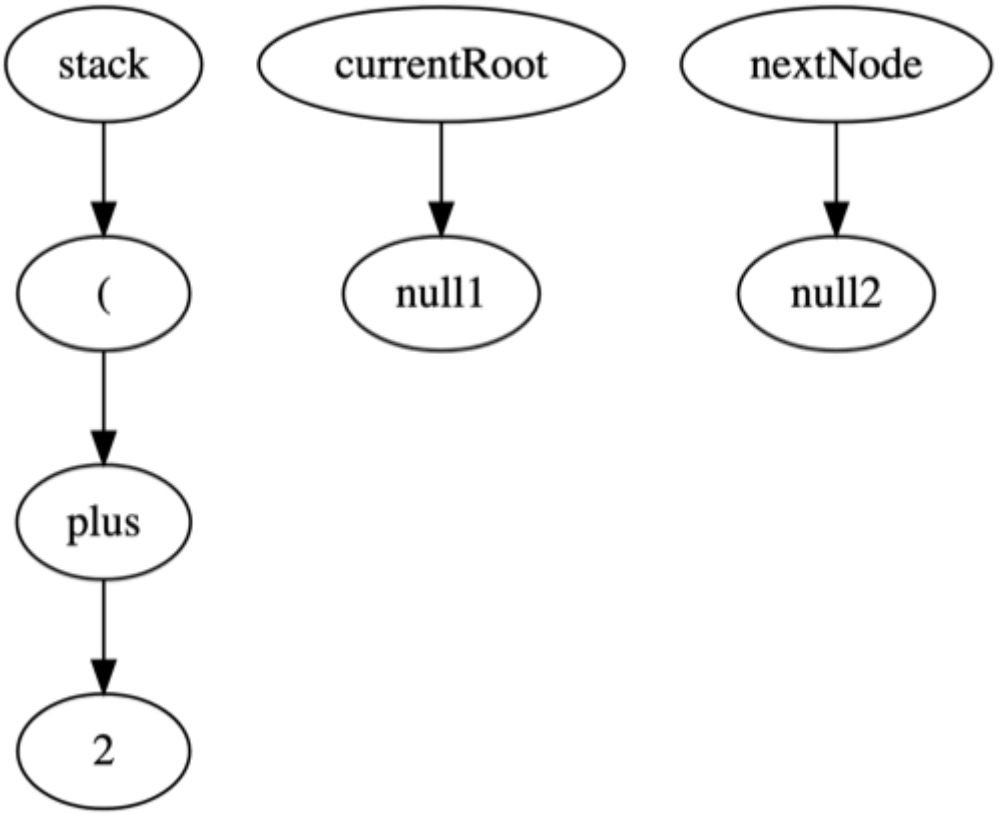

stack
currentRoot
nextNode
null1
plus
null2
2

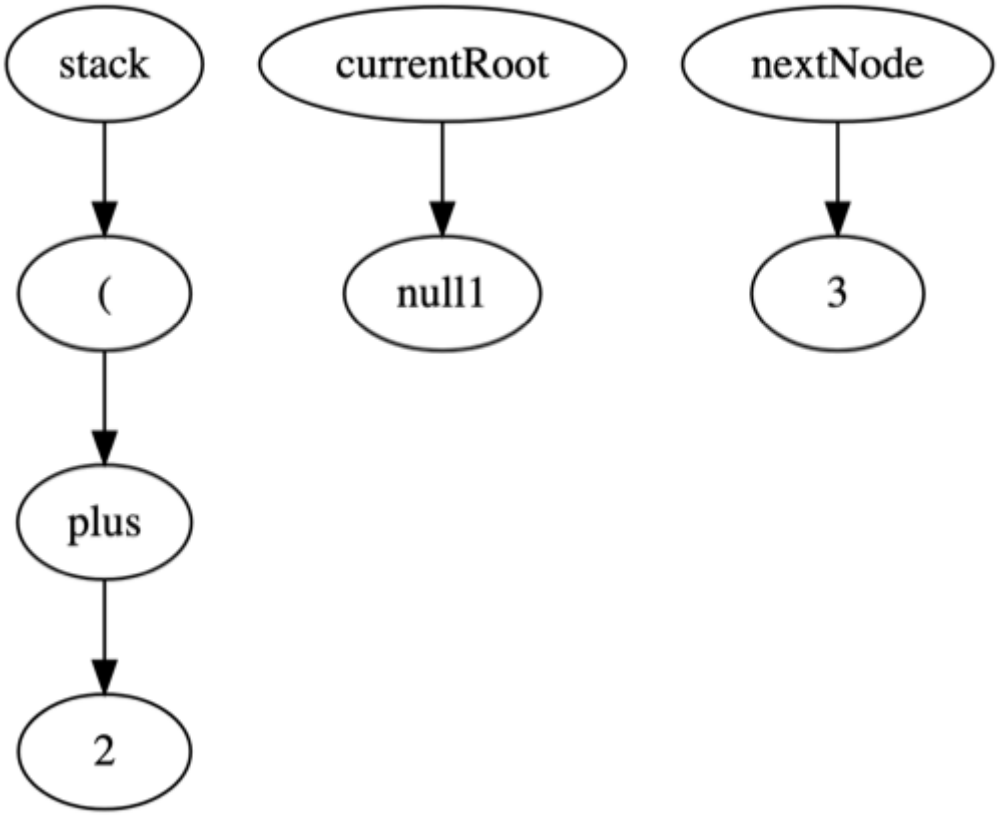

stack
currentRoot
nextNode
null1
plus
(
2

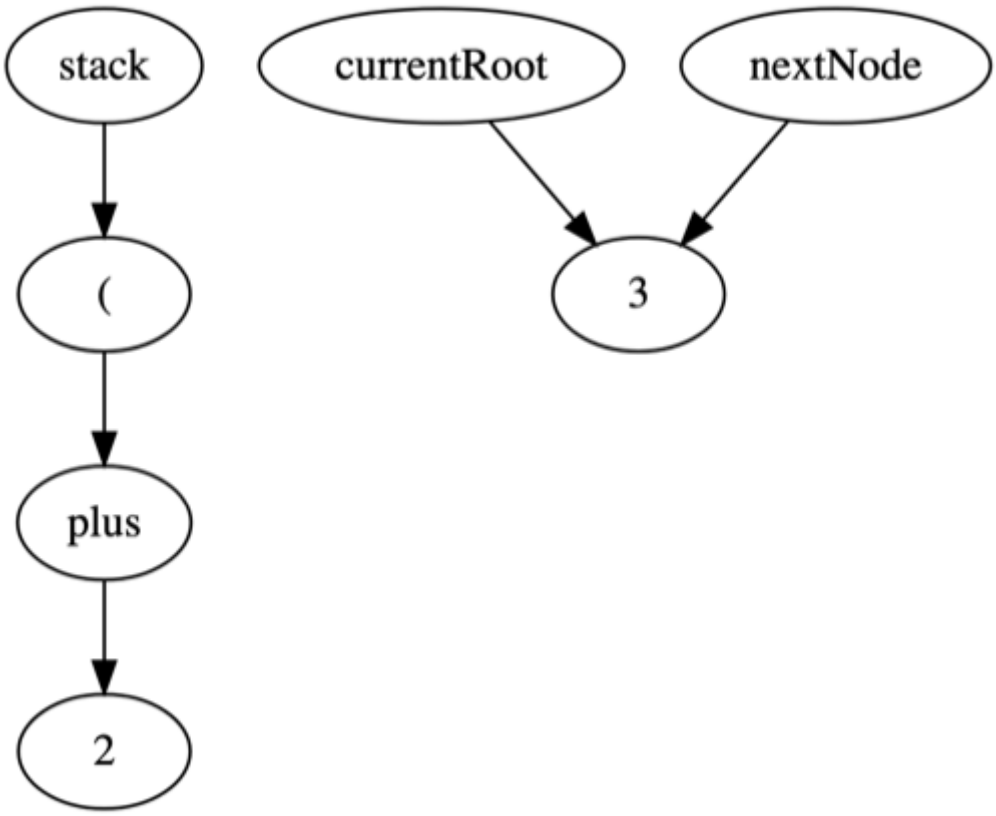

stack
currentRoot
nextNode
plus
(
2

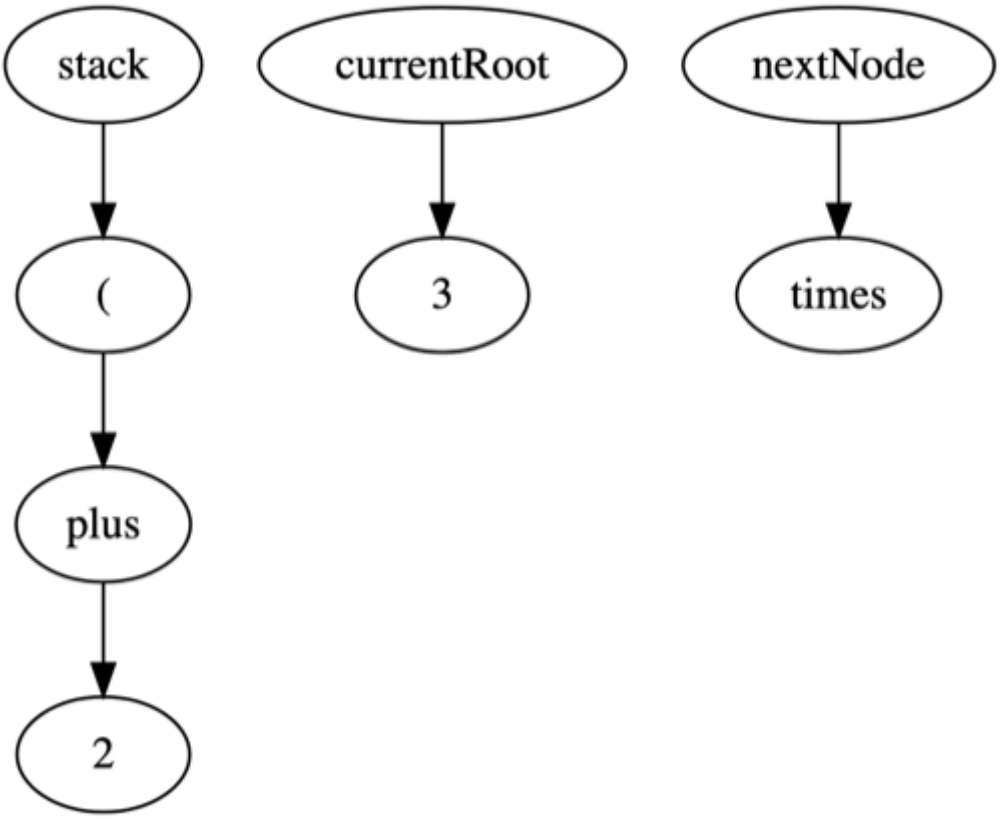

stack
currentRoot
nextNode
plus
null1
(
2

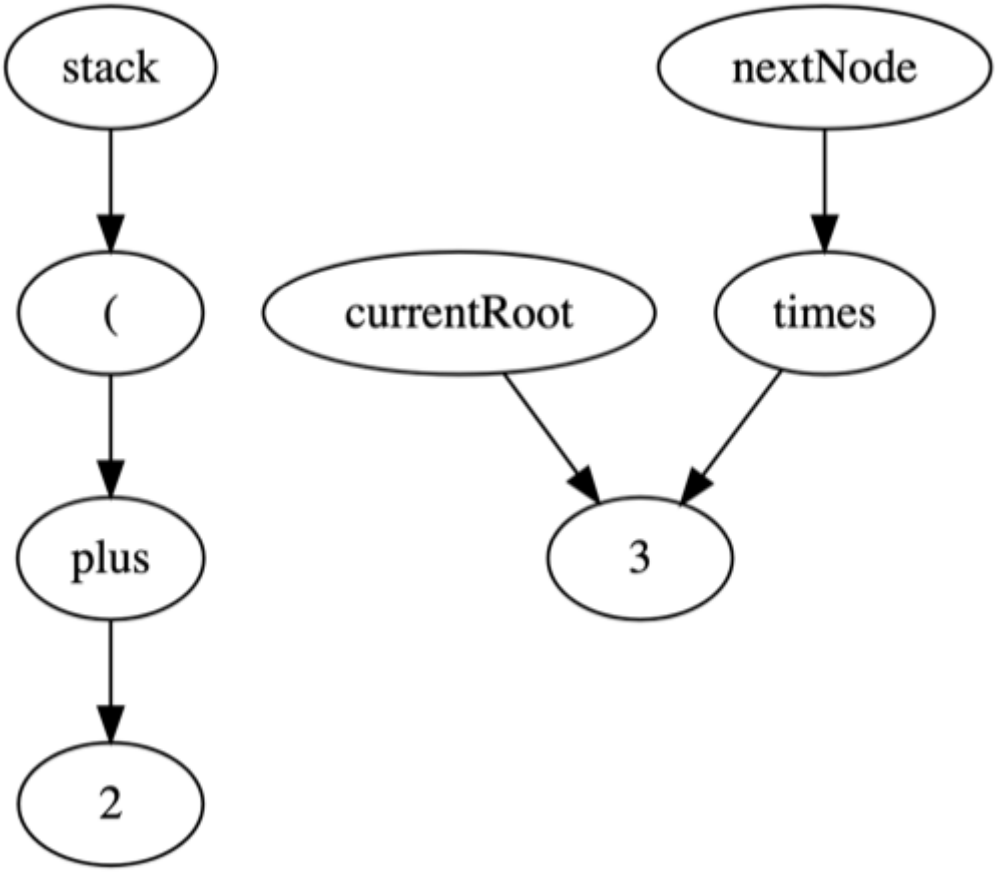

stack
nextNode
currentRoot
(
null1
plus
2

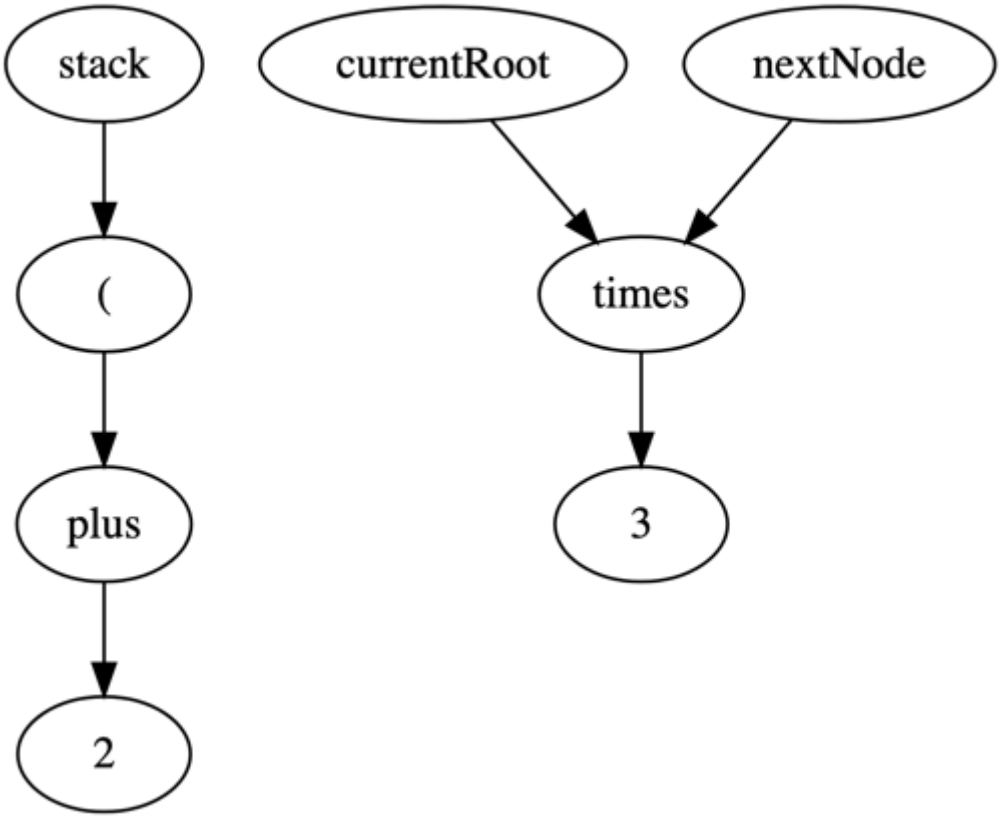

stack
currentRoot
nextNode
(
null1
null2
plus
2

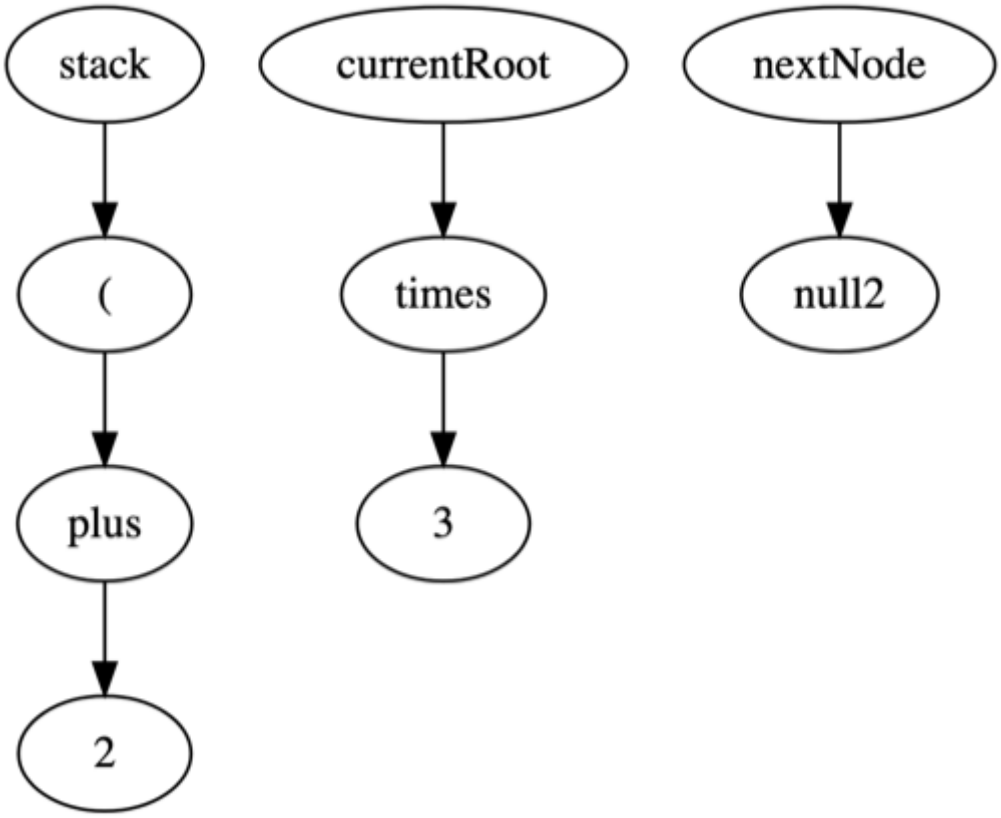

stack
currentRoot
nextNode
(
null1
3
plus
2

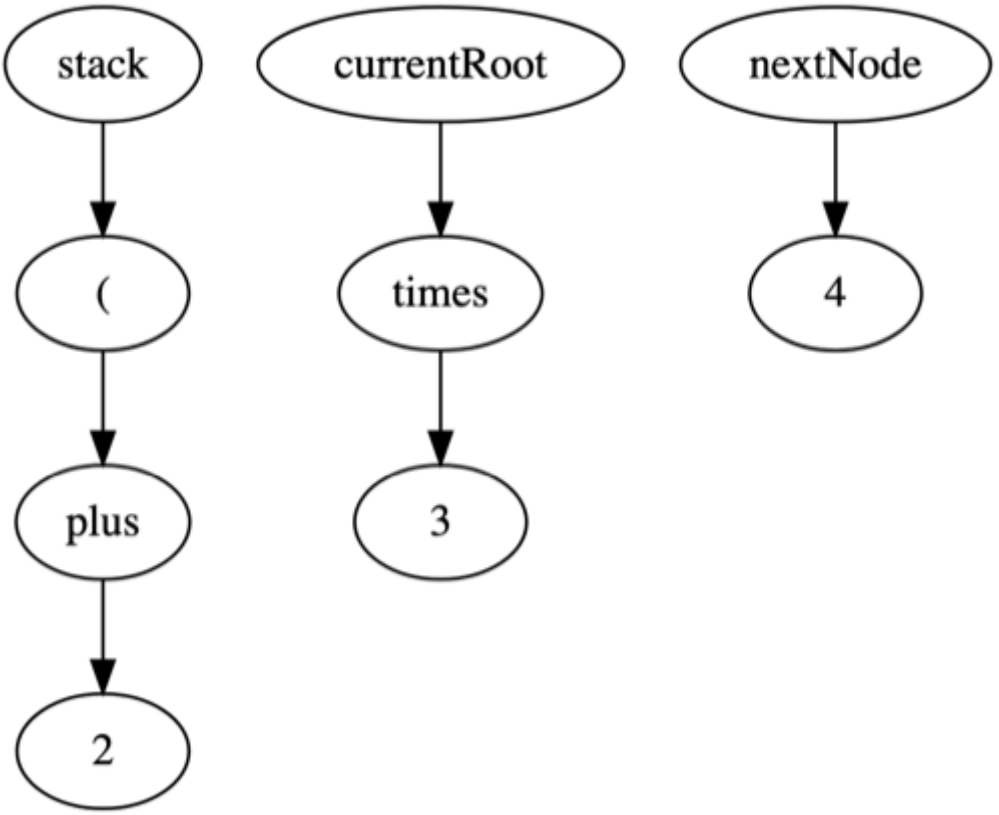

stack
currentRoot
nextNode
(
3
plus
2

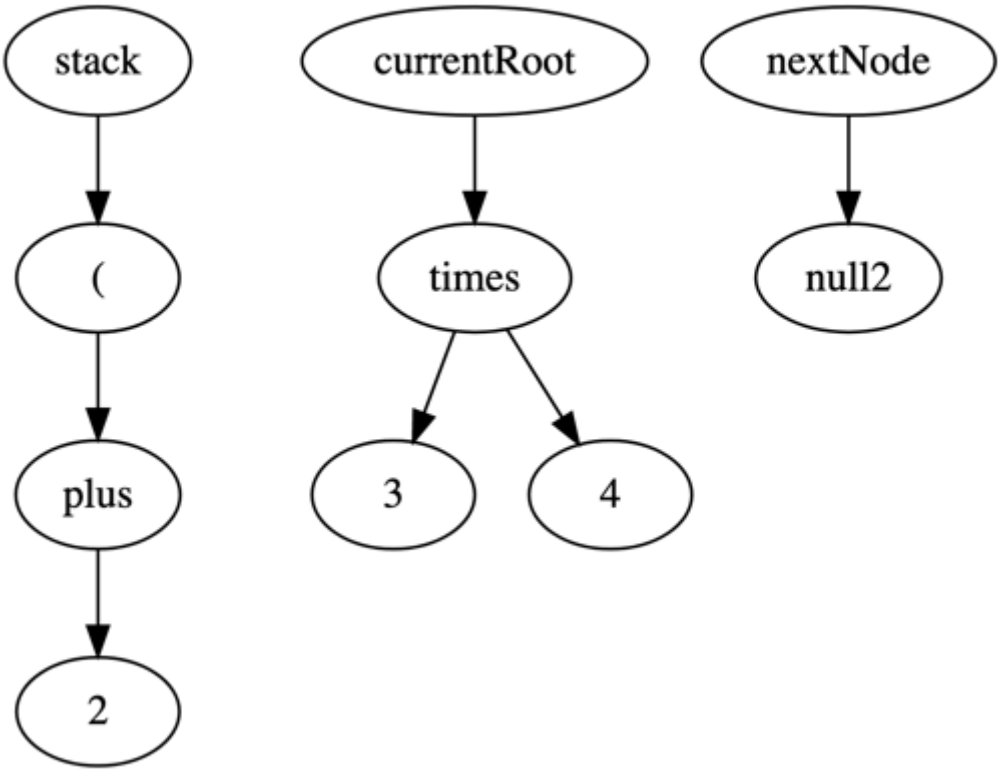

stack
currentRoot
nextNode
(
3
times
plus
2

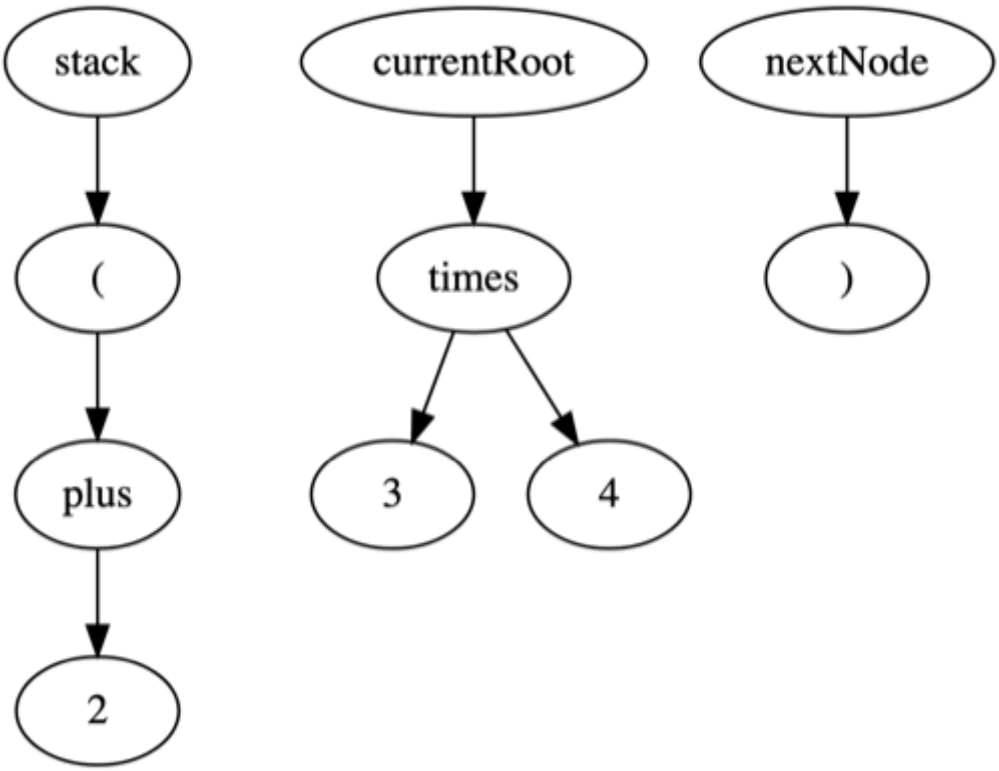

stack
nextNode
(
currentRoot
times
plus
3
2

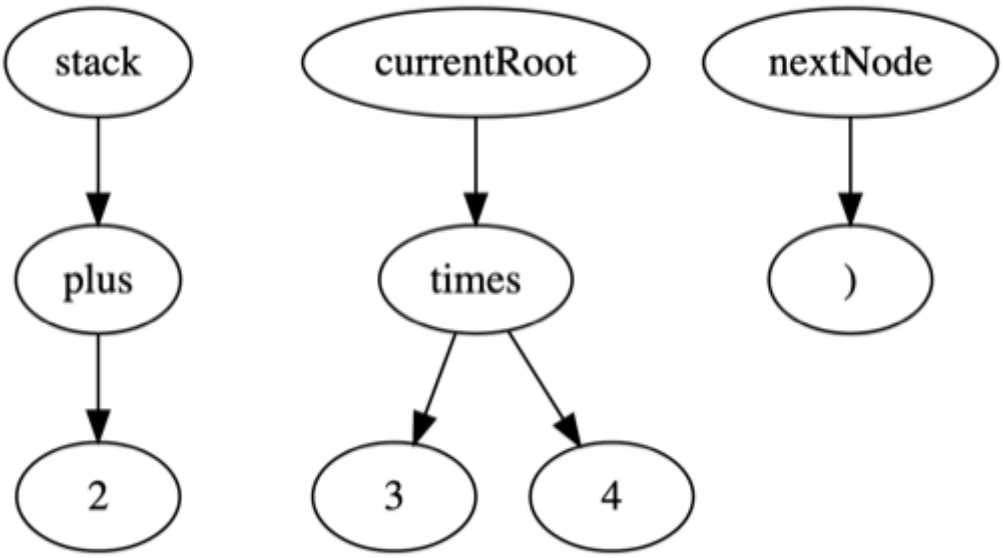

stack
currentRoot
nextNode
(
times
plus
3
2

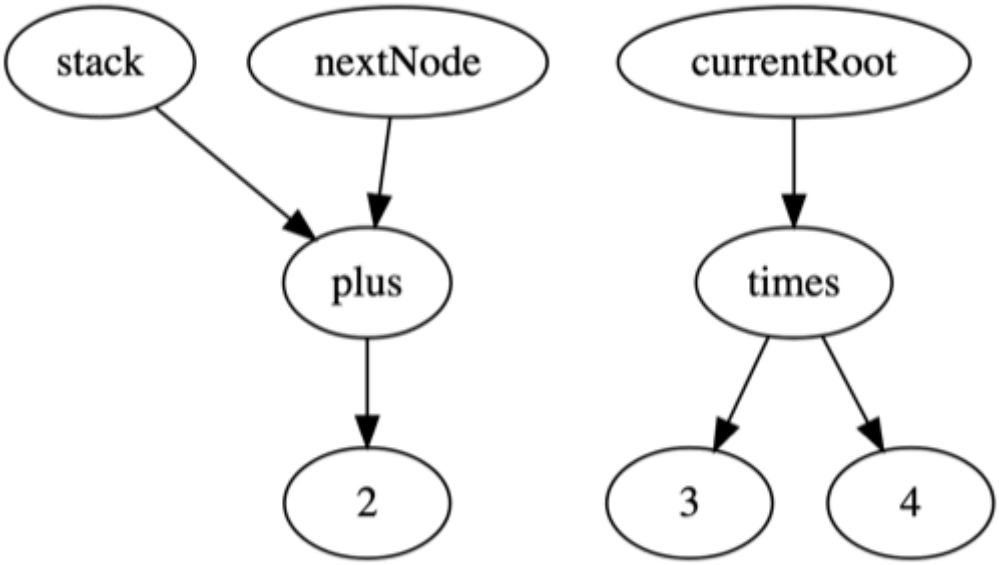

stack
currentRoot
nextNode
(
times
null2
plus
3
2

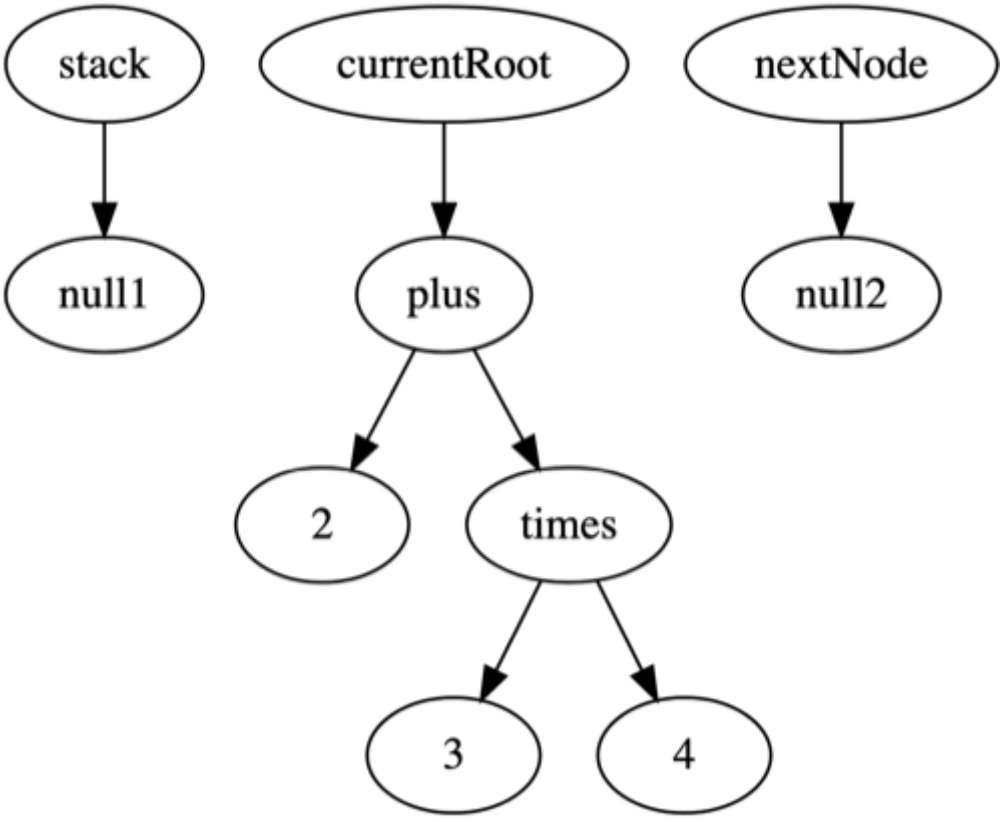

stack
currentRoot
nextNode
(
times
4
plus
3
2

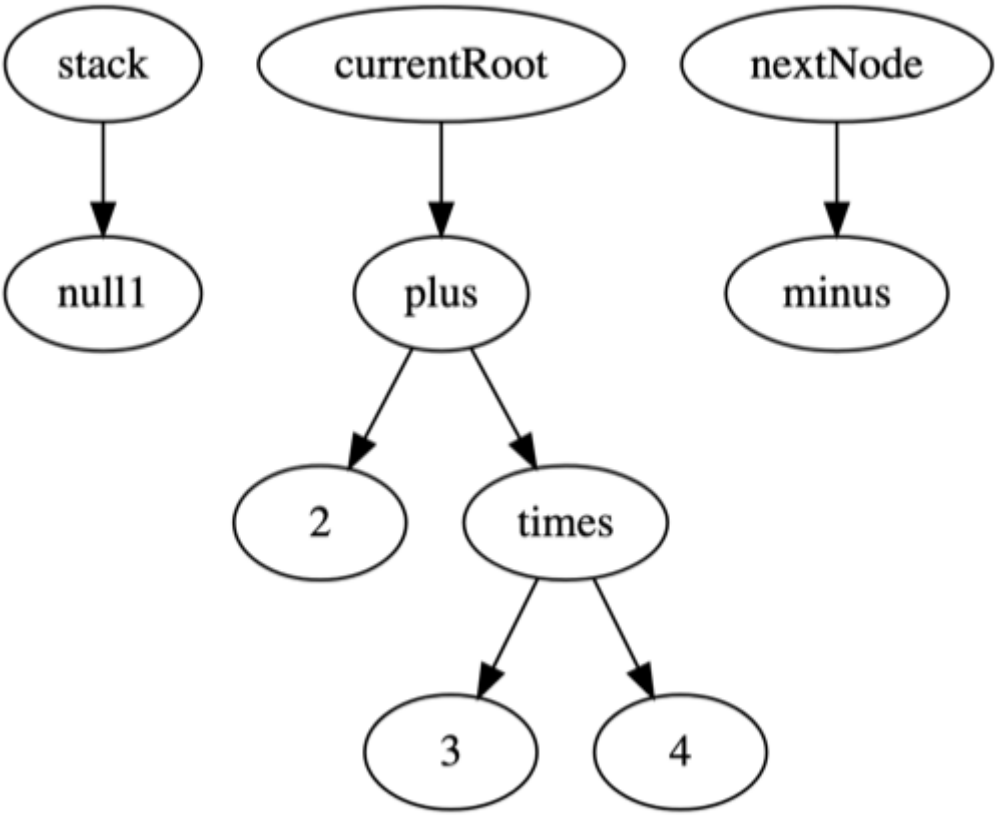

stack
currentRoot
nextNode
(
times
null2
plus
3
4
2

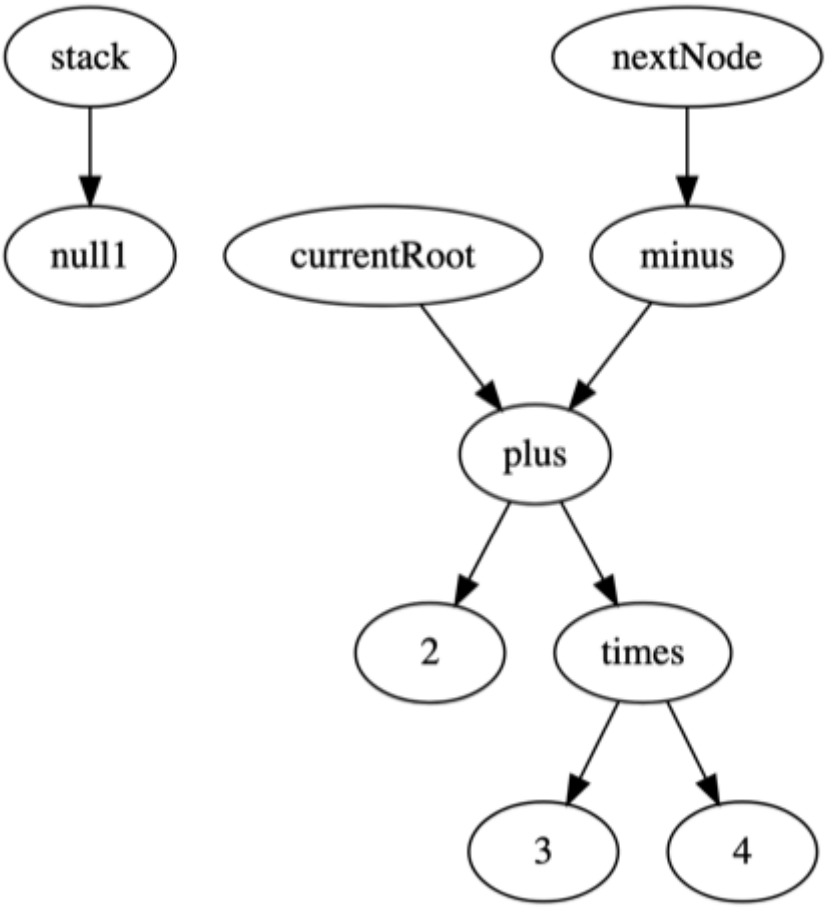

stack
currentRoot
nextNode
(
times
)
plus
3
4
2

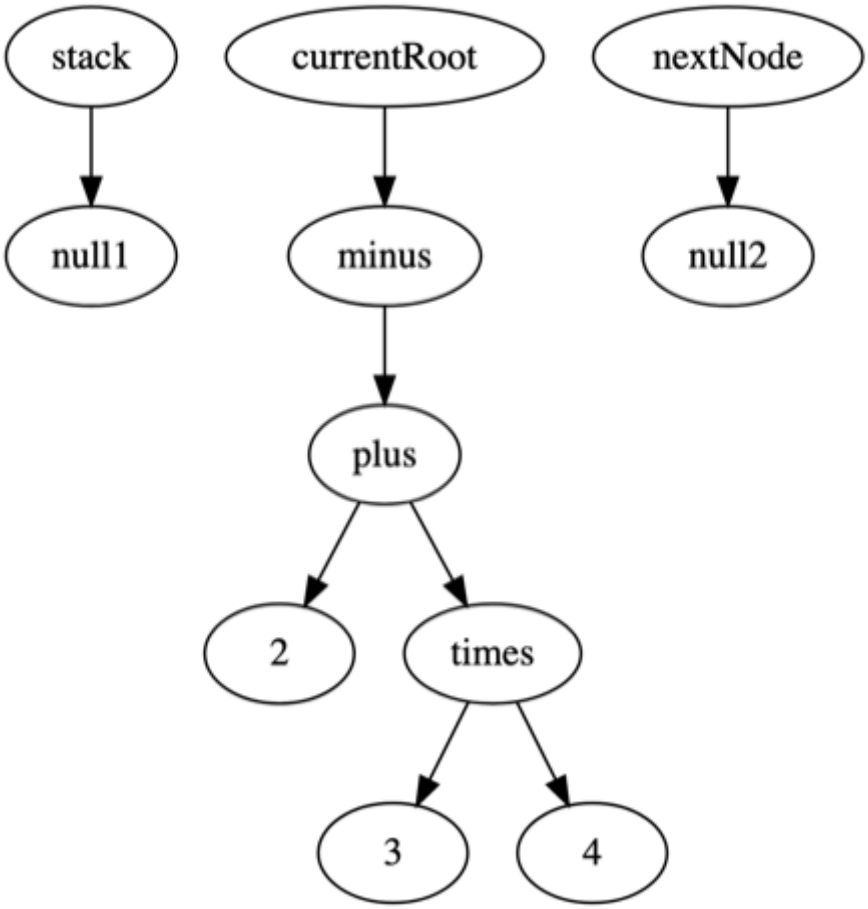

stack
currentRoot
nextNode
plus
times
)
2
3
4

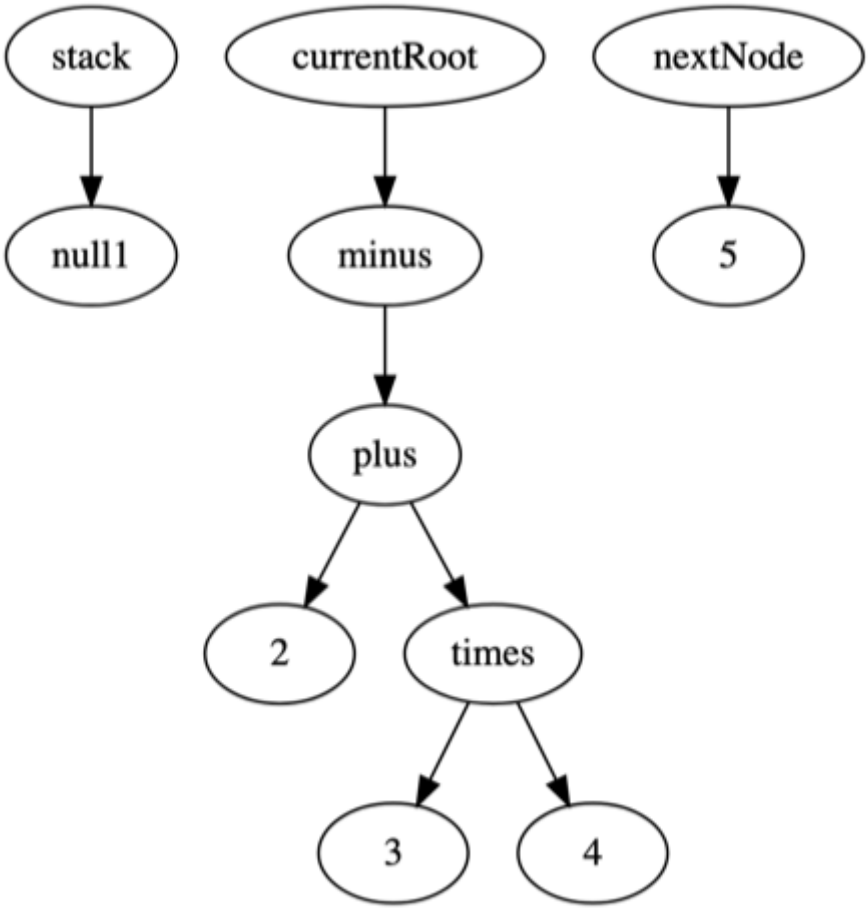

stack
nextNode
currentRoot
plus
times
2
3
4

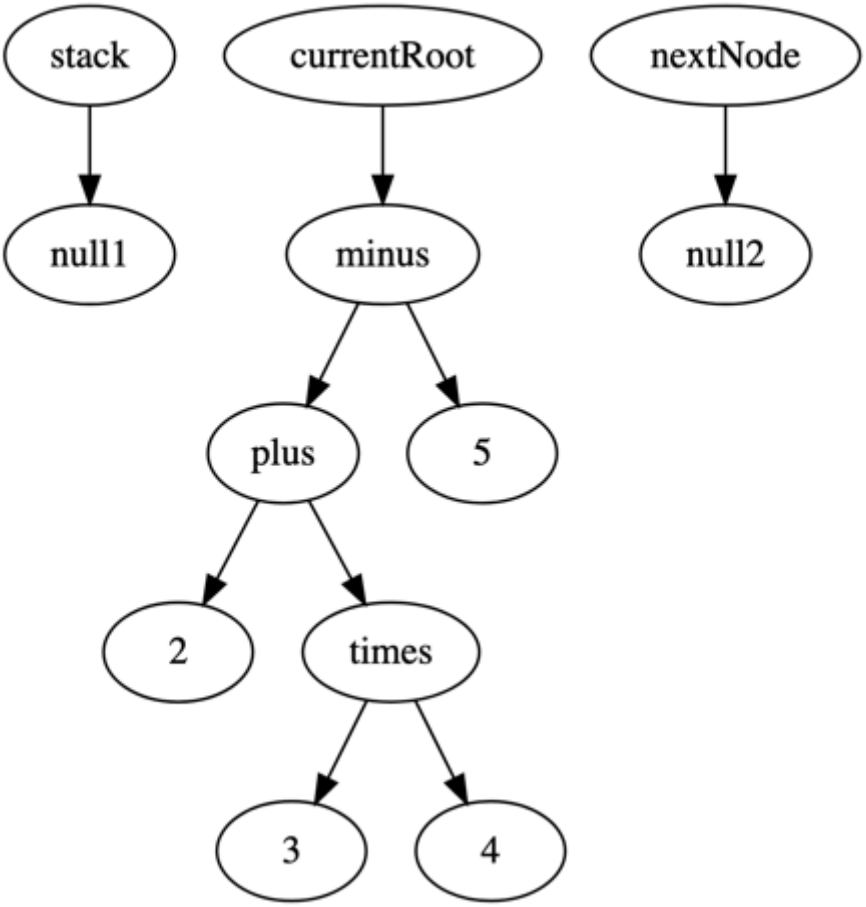

stack
currentRoot
nextNode
null1
plus
null2
2
times
3
4


stack
currentRoot
nextNode
null1
plus
minus
2
times
3
4


stack
nextNode
null1
currentRoot
minus
plus
2
times
3
4


stack
currentRoot
nextNode
null1
minus
null2
plus
2
times
3
4


stack
currentRoot
nextNode
null1
minus
5
plus
2
times
3
4


stack
currentRoot
nextNode
null1
minus
null2
plus
5
2
times
3
4

Appendix C: Maintaining multiple GIT repositories

How this repository is maintained

The GIT repository of this workshop is maintained at 3 servers simulteneously and maintained in all places at the same time. This is possible due to the flexibility in configuring the repository using GIT. This section explains how to configure a GIT repository to maintain in in multiple servers at same time

Creating repository copies

When the first repository is created then more repositories can be created by creating empty repositories on other servers and adding them as remotes to first repository. Assuming that the initial repository is at `git@github.com:<username>/<projectname>.git` (SSH Protocol is assumed here, but can also be done with https protocol) another empty repository should be created on another server. Here `<username>` is the user name of the user account on server. `<projectname>` is the name of the repository on the server. Assuming the another repository is at `git@gitlab.com:<username>/<projectname>.git`. While creating another repository please choose not to create any README or LICENSE files so the repository will not have any initial commits.

```
git remote add gitlab git@gitlab.com:<username>/<projectname>.git ①
git push gitlab master ②
```

① Add the additional server as another remote of the same initial repository

② Push the current version to another server making both repositories same

Configuring GIT for simultaneous push

Addition of a new branch named `all` manually in the `.git/config` file ensures that we have a branch which can be used to push the commits to all the remote servers at the same time. Open the config file at `.git/config` location and add the following section to it at the end

```
[remote "all"]
    url = git@github.com:<username>/<projectname>.git
    url = git@gitlab.com:<username>/<projectname>.git ①
```

① You can keep adding all the remote servers that have been added to the repository here one below the other to ensure push happens to all those servers.

After this configuration when the user commits and pushes to this branch alias `all` the push happens for all servers.

```
git push all
```

This command will push the commits to all the servers mentioned in the `all` branch configuration

Appendix D: How this manual is created

The content of this manual is written using a framework named Asciidoctor

Asciidoctor is a fast, open source text processor and publishing toolchain for converting AsciiDoc content to HTML5, DocBook, PDF, and other formats. Asciidoctor is written in Ruby and runs on all major operating systems. The Asciidoctor project is hosted on GitHub.

Asciidoctor reads and parses text written in the AsciiDoc syntax, then feeds the parse tree to a set of built-in converters to produce HTML5, DocBook 5, and man(ual) page output.

Using the asciidoctor framework and its plugins the content of this manual source is converted into HTML5, EPUB and PDF document. The diarams which are written in PlantUML format are converted into images by using asciidoctor-diagram plugin

How to build the manual

```
./manual/dockerbuild.sh
```