**Задание**

**Вариант 9.**

Реализуйте модель документа, которая обеспечивает хранение всей истории

изменений значений своих атрибутов. Реализуйте программу, которая

позволяет:

1. Вносить изменения в документ

2. Фиксировать изменения значений (для сохранения текущих изменений)

3. Выполнять откат состояния документа к любым изменениям, сделанным

ранее Система должна быть расширяема по документам любой структуры.

**Проектное решение**

**Описание Spring**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***№*** | ***Класс*** | ***Роль*** | ***Общие данные*** | ***Как защищается*** | ***С кем взаимодействует*** |
| ***1*** | ***ClientGenerator*** | ***Поток-производитель запросов*** | ***счётчик remaining, глобальная очередь*** | ***локальные поля — без синхронизации; запись в RequestQueue.put() использует внутренние блокировки LinkedBlockingQueue*** | ***кладёт Request в RequestQueue; пишет статистику в StatsAggregator*** |
| ***2*** | ***ClientConfig*** | ***Не содержит потоков; просто DTO-настройка клиента*** | ***—*** | ***—*** | ***создаётся ClientConfigFactory, читается ClientGenerator*** |
| ***3*** | ***ClientConfigFactory*** | ***Генерирует уникальные конфиги; не держит состояние*** | ***—*** | ***локальные переменные*** | ***вызывается ConcurrencyEngine при запуске*** |
| ***4*** | ***ConcurrencyEngine*** | ***Координатор: стартует/глушит клиенты, очередь, пул*** | ***списки потоков; очередь; пул*** | ***поток UI создаёт, shutdown кладёт poison-pill*** | ***управляет ClientGenerator и WorkerPool*** |
| ***5*** | ***RequestQueue*** | ***Разделяемый буфер (producer/consumer)*** | ***внутр. LinkedBlockingQueue<Request>*** | ***блокирующая коллекция потокобезопасна; счётчики через StatsAggregator*** | ***ClientGenerator → put/offer, Worker ← take*** |
| ***6*** | ***Request*** | ***Модель данных; генерирует уникальный id*** | ***статический AtomicLong SEQ*** | ***атомарный инкремент*** | ***создаётся клиентом, читается воркером*** |
| ***7*** | ***Worker*** | ***Поток-потребитель; исполняет команды*** | ***Document, DocumentHistoryLogger, StatsAggregator*** | ***take() блокирует очередь; документ синхронизирован; history — synchronized*** | ***берёт из RequestQueue; выполняет DocumentCommand*** |
| ***8*** | ***WorkerPool*** | ***Управляет пулом Worker (FixedThreadPool)*** | ***сервис ExecutorService*** | ***JDK ExecutorService*** | ***создаётся ConcurrencyEngine; shutdown — graceful*** |
| ***9*** | ***StatsAggregator*** | ***Центральная статистика системы*** | ***карты клиентов/воркеров (ConcurrentHashMap), списки запросов (synchronizedList), счётчики (AtomicLong), volatile queueSize*** | ***потокобезопасные коллекции, атомики*** | ***вызывают очередь, клиенты, воркеры, GUI*** |
| ***10*** | ***ClientStat*** | ***Статистика одного клиента*** | ***три AtomicLong*** | ***атомики*** | ***заполняет ClientGenerator; читает GUI*** |
| ***11*** | ***WorkerStat*** | ***Статистика воркера: счётчик и флаг busy*** | ***AtomicLong consumed, AtomicLong busyUntil*** | ***атомики; сравнение по времени*** | ***заполняет Worker; читает GUI*** |
| ***12*** | ***DocumentHistoryLogger*** | ***Хранит стеки undo/redo, общий для всех воркеров*** | ***Stack undo/redo*** | ***все публичные методы synchronized*** | ***вызывают команды Undo/Redo/Revert*** |
| ***13*** | ***ChangTextCommand*** | ***Изменяет документ*** | ***Document*** | ***полагается на синхронизацию документа*** | ***исполняется в Worker*** |
| ***14*** | ***Undo/Redo/RevertCommand*** | ***Работа с history*** | ***Document + DocumentHistoryLogger*** | ***logger synchronized; document synchronized*** | ***исполняются в Worker*** |
| ***15*** | ***EncryptionDocumentDecorator*** | ***Оборачивает документ; потоков нет*** | ***вызовы к inner Document*** | ***опирается на защиту базового документа*** | ***вызывается воркером*** |
| ***16*** | ***LoggingDocumentDecorator*** | ***Логирует операции*** | ***LogCollector*** | ***CopyOnWriteArrayList потокобезопасна*** | ***вызывается воркером*** |
| ***17*** | ***LogCollector*** | ***Кольцевой буфер строк логов*** | ***CopyOnWriteArrayList<String>*** | ***коллекция потокобезопасна*** | ***пишет декоратор, читает GUI*** |
| ***18*** | ***QueueDashboard*** | ***Swing-панель, работает в EDT*** | ***только читает StatsAggregator*** | ***потокобезопасные структуры; периодический Timer*** | ***визуализирует данные*** |
| ***19*** | ***SystemMonitorGUI*** | ***Создаёт ConcurrencyEngine, содержит вкладки*** | ***—*** | ***EDT*** | ***управляет жизненным циклом*** |
| ***20*** | ***Main*** | ***Точка входа; запускает GUI*** | ***—*** | ***—*** | ***запускает SystemMonitorGUI*** |

На рисунке 1 приведена диаграмма классов.

Изображение выглядит как диаграмма, План, Технический чертеж, зарисовка

Содержимое, созданное искусственным интеллектом, может быть неверным.

Рисунок 1. – Диаграмма классов

**Приложение 1**

**Программный код**

DocumentFactory.java

package core.factory**;**import core.documents.Document**;**public interface DocumentFactory {  
 Document createDocument()**;**}

PdfDocumentFactory.java

package core.factory**;**import core.documents.Document**;**import core.documents.PdfDocument**;**public class PdfDocumentFactory implements DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new PdfDocument()**;** }  
}

SpreadsheetDocumentFactory.java

package core.factory**;**import core.documents.Document**;**import core.documents.SpreadsheetDocument**;**public class SpreadsheetDocumentFactory implements DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new SpreadsheetDocument()**;** }  
}

TextDocumentFactory.java

package core.factory**;**import core.documents.Document**;**import core.documents.TextDocument**;**public class TextDocumentFactory implements DocumentFactory {  
 @Override  
 public Document createDocument() {  
 return new TextDocument()**;** }  
}

LogCollector.java

package core.logger**;**import java.util.List**;**import java.util.concurrent.CopyOnWriteArrayList**;**public class LogCollector {  
  
 private static final LogCollector *INSTANCE* = new LogCollector()**;** private static final int *MAX\_LOGS* = 10\_000**;** // кольцевой буфер  
  
 private final CopyOnWriteArrayList<String> logs = new CopyOnWriteArrayList<>()**;** private LogCollector() {}  
  
 public static LogCollector getInstance() { return *INSTANCE***;** }  
  
 public void add(String message) {  
 if (logs.size() >= *MAX\_LOGS*) {  
 int trim = *MAX\_LOGS* / 10**;** logs.subList(0**,** trim).clear()**;** }  
 logs.add(message)**;** }  
  
 public List<String> getLogs() { return List.*copyOf*(logs)**;** }  
}

Document.java

package core.documents**;**import core.memento.DocumentHistoryLogger**;**import core.memento.DocumentMemento**;**import core.observer.DocumentObserver**;**import java.util.concurrent.CopyOnWriteArrayList**;**public abstract class Document {  
 protected DocumentHistoryLogger historyLogger**;** public void setHistoryLogger(DocumentHistoryLogger logger) {  
 this.historyLogger = logger**;** }  
  
 private final CopyOnWriteArrayList<DocumentObserver> observers = new CopyOnWriteArrayList<>()**;** public abstract String getContent()**;** public abstract void setContent(String content)**;** public abstract DocumentMemento createMemento()**;** public abstract void restoreFromMemento(DocumentMemento memento)**;** public void addObserver(DocumentObserver observer) {  
 if (observer != null) observers.addIfAbsent(observer)**;** }  
  
 public void removeObserver(DocumentObserver observer) {  
 observers.remove(observer)**;** }  
  
 protected void notifyObservers() {  
 for (DocumentObserver obs : observers) obs.update(this)**;** }  
}

PdfDocument.java

package core.documents**;**import core.memento.DocumentMemento**;**import core.memento.GenericDocumentMemento**;**import java.util.Map**;**public class PdfDocument extends Document {  
  
 private volatile String content = ""**;** @Override  
 public synchronized String getContent() {  
 return content**;** }  
  
 @Override  
 public synchronized void setContent(String content) {  
 if (content == null) content = ""**;** if (!this.content.equals(content)) {  
 this.content = content**;** notifyObservers()**;** if (historyLogger != null)  
 historyLogger.addMemento(createMemento())**;** }  
 }  
  
 @Override  
 public DocumentMemento createMemento() {  
 return new GenericDocumentMemento(Map.*of*("content"**,** content))**;** }  
  
 @Override  
 public void restoreFromMemento(DocumentMemento memento) {  
 String restored = (String) memento.getState().getOrDefault("content"**,** "")**;** synchronized (this) {  
 if (!content.equals(restored)) {  
 content = restored**;** notifyObservers()**;** }  
 }  
 }  
}

SpreadsheetDocument.java

package core.documents**;**import core.memento.DocumentMemento**;**import core.memento.GenericDocumentMemento**;**import java.util.Map**;**public class SpreadsheetDocument extends Document {  
  
 private volatile String content = ""**;** @Override  
 public synchronized String getContent() {  
 return content**;** }  
  
 @Override  
 public synchronized void setContent(String content) {  
 if (content == null) content = ""**;** if (!this.content.equals(content)) {  
 this.content = content**;** notifyObservers()**;** if (historyLogger != null)  
 historyLogger.addMemento(createMemento())**;** }  
 }  
  
 @Override  
 public DocumentMemento createMemento() {  
 return new GenericDocumentMemento(Map.*of*("content"**,** content))**;** }  
  
 @Override  
 public void restoreFromMemento(DocumentMemento memento) {  
 String restored = (String) memento.getState().getOrDefault("content"**,** "")**;** synchronized (this) {  
 if (!content.equals(restored)) {  
 content = restored**;** notifyObservers()**;** }  
 }  
 }  
}

TextDocument.java

package core.documents**;**import core.memento.DocumentMemento**;**import core.memento.GenericDocumentMemento**;**import java.util.Map**;**public class TextDocument extends Document {  
  
 private volatile String content = ""**;** @Override  
 public synchronized String getContent() {  
 return content**;** }  
  
 @Override  
 public synchronized void setContent(String content) {  
 if (content == null) content = ""**;** if (!this.content.equals(content)) {  
 this.content = content**;** notifyObservers()**;** if (historyLogger != null)  
 historyLogger.addMemento(createMemento())**;** }  
 }  
  
 @Override  
 public DocumentMemento createMemento() {  
 return new GenericDocumentMemento(Map.*of*("content"**,** content))**;** }  
  
 @Override  
 public void restoreFromMemento(DocumentMemento memento) {  
 String restored = (String) memento.getState().getOrDefault("content"**,** "")**;** synchronized (this) {  
 if (!content.equals(restored)) {  
 content = restored**;** notifyObservers()**;** }  
 }  
 }  
}

DocumentDecorator.java

package core.decorators**;**import core.documents.Document**;**import core.memento.DocumentMemento**;**import core.observer.DocumentObserver**;**public abstract class DocumentDecorator extends Document {  
 protected Document document**;** public DocumentDecorator(Document document) {  
 if (document == null) {  
 throw new IllegalArgumentException("Оборачиваемый документ не может быть null")**;** }  
 this.document = document**;** }  
  
 @Override  
 public String getContent() {  
 return document.getContent()**;** }  
  
 @Override  
 public void setContent(String content) {  
 document.setContent(content)**;** }  
  
 @Override  
 public DocumentMemento createMemento() {  
 return document.createMemento()**;** }  
  
 @Override  
 public void restoreFromMemento(DocumentMemento memento) {  
 document.restoreFromMemento(memento)**;** }  
  
 @Override  
 public void addObserver(DocumentObserver observer) {  
 document.addObserver(observer)**;** }  
  
 @Override  
 public void removeObserver(DocumentObserver observer) {  
 document.removeObserver(observer)**;** }  
}

EncryptionDocumentDecorator.java

package core.decorators**;**import core.documents.Document**;**public class EncryptionDocumentDecorator extends DocumentDecorator {  
  
 public EncryptionDocumentDecorator(Document document) {  
 super(document)**;** }  
  
 @Override  
 public void setContent(String content) {  
 String encrypted = encrypt(content)**;** super.setContent(encrypted)**;** }  
  
 @Override  
 public String getContent() {  
 String encrypted = super.getContent()**;** return decrypt(encrypted)**;** }  
  
 private String encrypt(String input) {  
 if (input == null) return null**;** System.*out*.println("EncryptionDecorator: Шифрование данных...")**;** return new StringBuilder(input).reverse().toString()**;** }  
  
 private String decrypt(String input) {  
 if (input == null) return null**;** System.*out*.println("EncryptionDecorator: Дешифрование данных...")**;** return new StringBuilder(input).reverse().toString()**;** }  
}

LoggingDocumentDecorator.java

package core.decorators**;**import core.documents.Document**;**import core.logger.LogCollector**;**import core.memento.DocumentMemento**;**import core.observer.DocumentObserver**;**public class LoggingDocumentDecorator extends DocumentDecorator {  
 public LoggingDocumentDecorator(Document document) {  
 super(document)**;** }  
  
 @Override  
 public String getContent() {  
 LogCollector.*getInstance*().add("GET content")**;** return super.getContent()**;** }  
  
 @Override  
 public void setContent(String content) {  
 LogCollector.*getInstance*().add("SET content")**;** super.setContent(content)**;** }  
  
 @Override  
 public DocumentMemento createMemento() {  
 LogCollector.*getInstance*().add("CREATE core.memento")**;** return super.createMemento()**;** }  
  
 @Override  
 public void restoreFromMemento(DocumentMemento memento) {  
 LogCollector.*getInstance*().add("RESTORE from core.memento")**;** super.restoreFromMemento(memento)**;** }  
  
 @Override  
 public void addObserver(DocumentObserver observer) {  
 LogCollector.*getInstance*().add("ADD core.observer")**;** super.addObserver(observer)**;** }  
  
 @Override  
 public void removeObserver(DocumentObserver observer) {  
 LogCollector.*getInstance*().add("REMOVE core.observer")**;** super.removeObserver(observer)**;** }  
}

ConsoleLoggerObserver.java

package core.observer**;**import core.documents.Document**;**public class ConsoleLoggerObserver implements DocumentObserver {  
 @Override  
 public void update(Document document) {  
 System.*out*.println("ConsoleLoggerObserver: " + document.getContent())**;** }  
}

DocumentObserver.java

package core.observer**;**import core.documents.Document**;**public interface DocumentObserver {  
 void update(Document document)**;**}

DocumentMemento.java

package core.memento**;**import java.io.Serializable**;**import java.util.Map**;**public interface DocumentMemento extends Serializable {  
 Map<String**,** Object> getState()**;**}

GenericDocumentMemento.java

package core.memento**;**import java.io.Serial**;**import java.util.Map**;**public class GenericDocumentMemento implements DocumentMemento {  
  
 @Serial  
 private static final long *serialVersionUID* = 1L**;** private final Map<String**,** Object> state**;** public GenericDocumentMemento(Map<String**,** Object> state) {  
 this.state = Map.*copyOf*(state)**;** }  
  
 @Override  
 public Map<String**,** Object> getState() {  
 return state**;** }  
}

DocumentHistoryLogger.java

package core.memento**;**import java.io.\***;**import java.util.ArrayList**;**import java.util.List**;**import java.util.Stack**;**public class DocumentHistoryLogger {  
  
 private final Stack<DocumentMemento> undoStack = new Stack<>()**;** private final Stack<DocumentMemento> redoStack = new Stack<>()**;** public synchronized void addMemento(DocumentMemento memento) {  
 if (memento == null) return**;** undoStack.push(memento)**;** redoStack.clear()**;** }  
  
 public synchronized boolean canUndo() {  
 return !undoStack.isEmpty()**;** }  
  
 public synchronized boolean canRedo() {  
 return !redoStack.isEmpty()**;** }  
  
 public synchronized DocumentMemento undo(DocumentMemento current) {  
 if (!canUndo()) return null**;** if (current != null) redoStack.push(current)**;** return undoStack.pop()**;** }  
  
 public synchronized DocumentMemento redo(DocumentMemento current) {  
 if (!canRedo()) return null**;** if (current != null) undoStack.push(current)**;** return redoStack.pop()**;** }  
  
 public synchronized List<DocumentMemento> getHistory() {  
 return new ArrayList<>(undoStack)**;** }  
  
 public synchronized DocumentMemento getMementoAt(int index) {  
 if (index < 0 || index >= undoStack.size()) return null**;** return undoStack.get(index)**;** }  
  
 public synchronized void clearRedo() {  
 redoStack.clear()**;** }  
  
 public synchronized void saveHistoryToFile(File file) throws IOException {  
 try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(file))) {  
 oos.writeObject(new ArrayList<>(undoStack))**;** }  
 }  
  
 @SuppressWarnings("unchecked")  
 public synchronized void loadHistoryFromFile(File file) throws IOException**,** ClassNotFoundException {  
 try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(file))) {  
 List<DocumentMemento> list = (List<DocumentMemento>) ois.readObject()**;** undoStack.clear()**;** redoStack.clear()**;** undoStack.addAll(list)**;** }  
 }  
}

DocumentCommand.java

package core.commands**;**public interface DocumentCommand {  
 void execute()**;** void undo()**;**}

ChangeTextCommand.java

package core.commands**;**import core.documents.Document**;**public class ChangeTextCommand implements DocumentCommand {  
 private Document document**;** private String newText**;** private String previousText**;** public ChangeTextCommand(Document document**,** String newText) {  
 this.document = document**;** this.newText = newText**;** }  
  
 @Override  
 public void execute() {  
 previousText = document.getContent()**;** document.setContent(newText)**;** }  
  
 @Override  
 public void undo() {  
 document.setContent(previousText)**;** }  
}

UndoCommand.java

package core.commands**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**import core.memento.DocumentMemento**;**public class UndoCommand implements DocumentCommand {  
  
 private final Document document**;** private final DocumentHistoryLogger logger**;** public UndoCommand(Document document**,** DocumentHistoryLogger logger) {  
 this.document = document**;** this.logger = logger**;** }  
  
 @Override  
 public void execute() {  
 DocumentMemento current = document.createMemento()**;** DocumentMemento target = logger.undo(current)**;** if (target != null) document.restoreFromMemento(target)**;** }  
  
 @Override  
 public void undo() {  
 DocumentMemento current = document.createMemento()**;** DocumentMemento target = logger.redo(current)**;** if (target != null) document.restoreFromMemento(target)**;** }  
}

RedoCommand.java

package core.commands**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**import core.memento.DocumentMemento**;**public class RedoCommand implements DocumentCommand {  
  
 private final Document document**;** private final DocumentHistoryLogger logger**;** public RedoCommand(Document document**,** DocumentHistoryLogger logger) {  
 this.document = document**;** this.logger = logger**;** }  
  
 @Override  
 public void execute() {  
 DocumentMemento current = document.createMemento()**;** DocumentMemento target = logger.redo(current)**;** if (target != null) document.restoreFromMemento(target)**;** }  
  
 @Override  
 public void undo() {  
 DocumentMemento current = document.createMemento()**;** DocumentMemento target = logger.undo(current)**;** if (target != null) document.restoreFromMemento(target)**;** }  
}

RevertCommand.java

package core.commands**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**import core.memento.DocumentMemento**;**public class RevertCommand implements DocumentCommand {  
  
 private final Document document**;** private final DocumentHistoryLogger historyLogger**;** private DocumentMemento savedCurrent**;** public RevertCommand(Document document**,** DocumentHistoryLogger historyLogger) {  
 this.document = document**;** this.historyLogger = historyLogger**;** }  
  
 @Override  
 public void execute() {  
 savedCurrent = document.createMemento()**;** DocumentMemento target = historyLogger.undo(savedCurrent)**;** if (target != null) document.restoreFromMemento(target)**;** }  
  
 @Override  
 public void undo() {  
 if (savedCurrent == null) return**;** DocumentMemento back = document.createMemento()**;** historyLogger.addMemento(back)**;** document.restoreFromMemento(savedCurrent)**;** savedCurrent = null**;** }  
}

ClientConfig.java

package concurrency.client**;**import concurrency.model.RequestType**;**public record ClientConfig(  
 int id**,** RequestType[] allowedTypes**,** long baseDelayMs**,** long jitterMs**,** int burst  
) { }

ClientConfigFactory.java

package concurrency.client**;**import concurrency.model.RequestType**;**import java.util.LinkedHashSet**;**import java.util.Set**;**import java.util.concurrent.ThreadLocalRandom**;**public final class ClientConfigFactory {  
 private ClientConfigFactory() {}  
 public static ClientConfig random(int id) {  
 ThreadLocalRandom r = ThreadLocalRandom.*current*()**;** RequestType[] pool = {  
 RequestType.*CHANGE\_TEXT***,** RequestType.*UNDO***,** RequestType.*REDO***,** RequestType.*REVERT* }**;** int n = 1 + r.nextInt(pool.length)**;** Set<RequestType> chosen = new LinkedHashSet<>()**;** while (chosen.size() < n) {  
 chosen.add(pool[r.nextInt(pool.length)])**;** }  
 long baseDelay = 200 + r.nextLong(601)**;** long jitter = 100 + r.nextLong(401)**;** int burst = 1 + r.nextInt(4)**;** return new ClientConfig(id**,** chosen.toArray(RequestType[]::new)**,** baseDelay**,** jitter**,** burst)**;** }  
}

ClientGenerator.java

package concurrency.client**;**import concurrency.model.\***;**import concurrency.queue.RequestQueue**;**import concurrency.monitor.StatsAggregator**;**import java.util.concurrent.ThreadLocalRandom**;**import java.util.concurrent.atomic.AtomicInteger**;**public class ClientGenerator implements Runnable {  
  
 private static final AtomicInteger *TXT* = new AtomicInteger()**;** private final int id**;** private final ClientConfig cfg**;** private final RequestQueue q**;** private final StatsAggregator agg**;** private int remaining**;** private volatile boolean run = true**;** private long lastSent = System.*currentTimeMillis*()**;** public ClientGenerator(int id**,** ClientConfig cfg**,** int quota**,** RequestQueue q**,** StatsAggregator agg) {  
 this.id = id**;** this.cfg = cfg**;** this.remaining = quota**;** this.q = q**;** this.agg = agg**;** }  
  
 public void stop() {  
 run = false**;** }  
  
 @Override  
 public void run() {  
 var rnd = ThreadLocalRandom.*current*()**;** var stat = agg.client(id)**;** while (run && remaining > 0) {  
 try {  
 for (int i = 0**;** i < cfg.burst() && run && remaining > 0**;** i++) {  
 RequestType t = pick(rnd)**;** String c = t == RequestType.*CHANGE\_TEXT* ? "txt-" + id + "-" + *TXT*.incrementAndGet() : null**;** q.put(new Request(t**,** c))**;** remaining--**;** long now = System.*currentTimeMillis*()**;** stat.addDelay(now - lastSent)**;** stat.incProduced()**;** lastSent = now**;** }  
 long d = cfg.baseDelayMs() + rnd.nextLong(cfg.jitterMs())**;** Thread.*sleep*(d)**;** } catch (InterruptedException ie) {  
 Thread.*currentThread*().interrupt()**;** break**;** }  
 }  
 }  
  
 private RequestType pick(ThreadLocalRandom r) {  
 var arr = cfg.allowedTypes()**;** return arr[r.nextInt(arr.length)]**;** }  
}

ConcurrencyEngine.java

package concurrency.engine**;**import concurrency.client.ClientConfig**;**import concurrency.client.ClientConfigFactory**;**import concurrency.client.ClientGenerator**;**import concurrency.monitor.StatsAggregator**;**import concurrency.model.Request**;**import concurrency.model.RequestType**;**import concurrency.queue.RequestQueue**;**import concurrency.server.WorkerPool**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**import java.util.ArrayList**;**import java.util.List**;**import java.util.concurrent.TimeUnit**;**public class ConcurrencyEngine {  
  
 private final StatsAggregator agg**;** private final RequestQueue queue**;** private final WorkerPool workers**;** private final List<ClientGenerator> gens = new ArrayList<>()**;** private final List<Thread> genThreads = new ArrayList<>()**;** public ConcurrencyEngine(int clients**,** int wcnt**,** int cap**,** int total**,** Document doc**,** DocumentHistoryLogger hist) {  
  
 agg = new StatsAggregator(cap)**;** queue = new RequestQueue(cap**,** agg)**;** int quota = total / clients**;** for (int i = 1**;** i <= clients**;** i++) {  
 ClientConfig cfg = ClientConfigFactory.*random*(i)**;** ClientGenerator g = new ClientGenerator(i**,** cfg**,** quota**,** queue**,** agg)**;** gens.add(g)**;** Thread t = new Thread(g**,** "Client-" + i)**;** genThreads.add(t)**;** t.start()**;** }  
  
 workers = new WorkerPool(wcnt**,** queue**,** doc**,** hist**,** agg)**;** }  
  
 public StatsAggregator stats() {  
 return agg**;** }  
  
 public void shutdown() {  
 int w = workers.size()**;** for (int i = 0**;** i < w**;** i++) {  
 try {  
 queue.offer(new Request(RequestType.*SHUTDOWN***,** null)**,** 2**,** TimeUnit.*SECONDS*)**;** } catch (InterruptedException ie) {  
 Thread.*currentThread*().interrupt()**;** }  
 }  
 gens.forEach(ClientGenerator::stop)**;** genThreads.forEach(Thread::interrupt)**;** for (Thread t : genThreads) {  
 try {  
 t.join(2000)**;** } catch (InterruptedException ignored) {}  
 }  
 workers.shutdownGracefully()**;** }  
}

Request.java

package concurrency.model**;**import java.util.concurrent.atomic.AtomicLong**;**public class Request {  
 private static final AtomicLong *SEQ* = new AtomicLong()**;** private final long id**;** private final RequestType type**;** private final String content**;** private final long createdAt**;** public Request(RequestType type**,** String content) {  
 this.id = *SEQ*.incrementAndGet()**;** this.type = type**;** this.content = content**;** this.createdAt = System.*currentTimeMillis*()**;** }  
  
 public long getId() { return id**;** }  
 public RequestType getType(){ return type**;** }  
 public String getContent() { return content**;** }  
 public long getCreatedAt() { return createdAt**;** }  
  
 @Override  
 public boolean equals(Object o) {  
 return o instanceof Request r && r.id == id**;** }  
  
 @Override  
 public int hashCode() {  
 return Long.*hashCode*(id)**;** }  
}

RequestType

package concurrency.model**;**public enum RequestType {  
 *CHANGE\_TEXT***,** *UNDO***,** *REDO***,** *REVERT***,** *SHUTDOWN*}

ClientStat.java

package concurrency.monitor**;**import java.util.concurrent.atomic.AtomicLong**;**public final class ClientStat {  
 private final AtomicLong produced = new AtomicLong()**;** private final AtomicLong totalDelay = new AtomicLong()**;** private final AtomicLong requests = new AtomicLong()**;** public void incProduced() { produced.incrementAndGet()**;** }  
 public void addDelay(long d) { totalDelay.addAndGet(d)**;** requests.incrementAndGet()**;** }  
 public long produced() { return produced.get()**;** }  
 public double avgDelay() { long r = requests.get()**;** return r == 0 ? 0 : (double) totalDelay.get() / r**;** }  
}

StatsAggregator.java

package concurrency.monitor**;**import concurrency.model.Request**;**import java.util.\***;**import java.util.concurrent.\***;**import java.util.concurrent.atomic.AtomicLong**;**public class StatsAggregator {  
  
 private final int capacity**;** private volatile int queueSize**;** private final ConcurrentMap<Integer**,** ClientStat> clients = new ConcurrentHashMap<>()**;** private final ConcurrentMap<Integer**,** WorkerStat> workers = new ConcurrentHashMap<>()**;** private final List<Request> allRequests = Collections.*synchronizedList*(new ArrayList<>())**;** private final ConcurrentMap<Long**,** Request> queued = new ConcurrentHashMap<>()**;** private final AtomicLong producedTotal = new AtomicLong()**;** private final AtomicLong consumedTotal = new AtomicLong()**;** public StatsAggregator(int capacity) {  
 this.capacity = capacity**;** }  
  
 public void setQueueSize(int size) {  
 queueSize = size**;** }  
  
 public int capacity() {  
 return capacity**;** }  
  
 public int queueSize() {  
 return queueSize**;** }  
  
 public void incProducedTotal() {  
 producedTotal.incrementAndGet()**;** }  
  
 public void incConsumedTotal() {  
 consumedTotal.incrementAndGet()**;** }  
  
 public long producedTotal() {  
 return producedTotal.get()**;** }  
  
 public long consumedTotal() {  
 return consumedTotal.get()**;** }  
  
 public ClientStat client(int id) {  
 return clients.computeIfAbsent(id**,** k -> new ClientStat())**;** }  
  
 public WorkerStat worker(int id) {  
 return workers.computeIfAbsent(id**,** k -> new WorkerStat())**;** }  
  
 public void enqueue(Request r) {  
 allRequests.add(r)**;** queued.put(r.getId()**,** r)**;** }  
  
 public void dequeue(Request r) {  
 queued.remove(r.getId())**;** }  
  
 public List<Request> allRequests() {  
 return List.*copyOf*(allRequests)**;** }  
  
 public List<Request> queuedRequests() {  
 return List.*copyOf*(queued.values())**;** }  
  
 public List<Object[]> clientRows() {  
 List<Object[]> rows = new ArrayList<>()**;** clients.forEach((id**,** st) ->  
 rows.add(new Object[]{id**,** st.produced()**,** String.*format*("%.0f"**,** st.avgDelay())}))**;** rows.sort(Comparator.*comparingLong*(o -> -(Long) o[1]))**;** return rows**;** }  
  
 public List<Object[]> workerRows() {  
 List<Object[]> rows = new ArrayList<>()**;** workers.forEach((id**,** st) ->  
 rows.add(new Object[]{id**,** st.consumed()**,** st.isBusy() ? "Busy" : "Idle"}))**;** rows.sort(Comparator.*comparingLong*(o -> -(Long) o[1]))**;** return rows**;** }  
}

WorkerStat.java

package concurrency.monitor**;**import java.util.concurrent.atomic.AtomicLong**;**public final class WorkerStat {  
 private final AtomicLong consumed = new AtomicLong()**;** private final AtomicLong busyUntil = new AtomicLong()**;** private static final long *BUSY\_WINDOW\_MS* = 150**;** public void markBusy() {  
 busyUntil.set(System.*currentTimeMillis*() + *BUSY\_WINDOW\_MS*)**;** }  
  
 public void incConsumed() {  
 consumed.incrementAndGet()**;** }  
  
 public long consumed() {  
 return consumed.get()**;** }  
  
 public boolean isBusy() {  
 return busyUntil.get() > System.*currentTimeMillis*()**;** }  
}

RequestQueue.java

package concurrency.queue**;**import concurrency.model.Request**;**import concurrency.monitor.StatsAggregator**;**import java.util.concurrent.BlockingQueue**;**import java.util.concurrent.LinkedBlockingQueue**;**import java.util.concurrent.TimeUnit**;**public class RequestQueue {  
  
 private final BlockingQueue<Request> queue**;** private final StatsAggregator agg**;** public RequestQueue(int capacity**,** StatsAggregator agg) {  
 this.queue = new LinkedBlockingQueue<>(capacity)**;** this.agg = agg**;** }  
  
 public void put(Request r) throws InterruptedException {  
 queue.put(r)**;** recordEnqueue(r)**;** }  
  
 public boolean offer(Request r**,** long timeout**,** TimeUnit unit) throws InterruptedException {  
 boolean ok = queue.offer(r**,** timeout**,** unit)**;** if (ok) recordEnqueue(r)**;** return ok**;** }  
  
 public Request take() throws InterruptedException {  
 Request r = queue.take()**;** agg.dequeue(r)**;** agg.setQueueSize(queue.size())**;** return r**;** }  
  
 public int size() { return queue.size()**;** }  
  
 private void recordEnqueue(Request r) {  
 agg.incProducedTotal()**;** agg.enqueue(r)**;** agg.setQueueSize(queue.size())**;** }  
}

Worker.java

package concurrency.server**;**import concurrency.model.Request**;**import concurrency.model.RequestType**;**import concurrency.queue.RequestQueue**;**import concurrency.monitor.StatsAggregator**;**import core.commands.\***;**import core.commands.DocumentCommand**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**public class Worker implements Runnable {  
  
 private final int id**;** private final RequestQueue q**;** private final Document doc**;** private final DocumentHistoryLogger hist**;** private final StatsAggregator agg**;** public Worker(int id**,** RequestQueue q**,** Document doc**,** DocumentHistoryLogger h**,** StatsAggregator a) {  
 this.id = id**;** this.q = q**;** this.doc = doc**;** this.hist = h**;** this.agg = a**;** }  
  
 @Override  
 public void run() {  
 var stat = agg.worker(id)**;** boolean shutdownReceived = false**;** while (true) {  
 try {  
 Request r = q.take()**;** stat.markBusy()**;** if (r.getType() == RequestType.*SHUTDOWN*) {  
 shutdownReceived = true**;** if (q.size() == 0) break**;** else continue**;** }  
 switch (r.getType()) {  
 case *CHANGE\_TEXT* -> new ChangeTextCommand(doc**,** r.getContent()).execute()**;** case *UNDO* -> new UndoCommand(doc**,** hist).execute()**;** case *REDO* -> new RedoCommand(doc**,** hist).execute()**;** case *REVERT* -> new RevertCommand(doc**,** hist).execute()**;** }  
 stat.incConsumed()**;** agg.incConsumedTotal()**;** if (shutdownReceived && q.size() == 0) break**;** } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt()**;** break**;** }  
 }  
 }  
}

WorkerPool.java

package concurrency.server**;**import concurrency.monitor.StatsAggregator**;**import concurrency.queue.RequestQueue**;**import core.documents.Document**;**import core.memento.DocumentHistoryLogger**;**import java.util.ArrayList**;**import java.util.List**;**import java.util.concurrent.ExecutorService**;**import java.util.concurrent.Executors**;**import java.util.concurrent.TimeUnit**;**public class WorkerPool {  
  
 private final ExecutorService exec**;** private final List<Worker> list = new ArrayList<>()**;** public WorkerPool(int size**,** RequestQueue q**,** Document d**,** DocumentHistoryLogger h**,** StatsAggregator a) {  
  
 exec = Executors.*newFixedThreadPool*(size)**;** for (int i = 1**;** i <= size**;** i++) {  
 Worker w = new Worker(i**,** q**,** d**,** h**,** a)**;** list.add(w)**;** exec.submit(w)**;** }  
 }  
  
 public void shutdownGracefully() {  
 exec.shutdown()**;** try {  
 if (!exec.awaitTermination(10**,** TimeUnit.*SECONDS*))  
 exec.shutdownNow()**;** } catch (InterruptedException ie) {  
 exec.shutdownNow()**;** Thread.*currentThread*().interrupt()**;** }  
 }  
  
 public int size() {  
 return list.size()**;** }  
}

LogsPanel.java

package gui**;**import core.logger.LogCollector**;**import javax.swing.\***;**import java.util.List**;**public class LogsPanel extends JScrollPane {  
  
 private final JTextArea area = new JTextArea()**;** public LogsPanel() {  
 setViewportView(area)**;** area.setEditable(false)**;** new Timer(500**,** e -> refresh()).start()**;** }  
  
 private void refresh() {  
 List<String> logs = LogCollector.*getInstance*().getLogs()**;** StringBuilder sb = new StringBuilder()**;** for (String s : logs) sb.append(s).append('\n')**;** area.setText(sb.toString())**;** area.setCaretPosition(area.getDocument().getLength())**;** }  
}

QueueDashboard.java

package gui**;**import concurrency.monitor.StatsAggregator**;**import concurrency.model.Request**;**import javax.swing.\***;**import javax.swing.table.DefaultTableModel**;**import java.awt.\***;**import java.util.List**;**public class QueueDashboard extends JPanel {  
  
 private final StatsAggregator agg**;** private final JProgressBar bar = new JProgressBar()**;** private final DefaultTableModel clientM = new DefaultTableModel(  
 new String[]{"Client"**,** "Produced"**,** "Avg delay"}**,** 0)**;** private final DefaultTableModel workerM = new DefaultTableModel(  
 new String[]{"Worker"**,** "Consumed"**,** "State"}**,** 0)**;** private final DefaultListModel<String> allM = new DefaultListModel<>()**;** public QueueDashboard(StatsAggregator agg) {  
 this.agg = agg**;** setLayout(new BorderLayout(8**,**8))**;** bar.setStringPainted(true)**;** add(bar**,** BorderLayout.*NORTH*)**;** JTable cl = new JTable(clientM)**;** JTable wk = new JTable(workerM)**;** JSplitPane tables = new JSplitPane(  
 JSplitPane.*HORIZONTAL\_SPLIT***,** new JScrollPane(cl)**,** new JScrollPane(wk))**;** tables.setResizeWeight(0.5)**;** JScrollPane allPane = new JScrollPane(new JList<>(allM))**;** allPane.setBorder(BorderFactory.*createTitledBorder*("Requests (all)"))**;** JSplitPane split = new JSplitPane(  
 JSplitPane.*HORIZONTAL\_SPLIT***,** tables**,** allPane)**;** split.setResizeWeight(0.7)**;** add(split**,** BorderLayout.*CENTER*)**;** new Timer(500**,** e -> refresh()).start()**;** }  
  
 private void refresh() {  
 bar.setMaximum(agg.producedTotal() == 0 ? 1 : (int) agg.producedTotal())**;** bar.setValue((int) agg.consumedTotal())**;** bar.setString(agg.consumedTotal() + " / " + agg.producedTotal())**;** *rebuild*(clientM**,** agg.clientRows())**;** *rebuild*(workerM**,** agg.workerRows())**;** *fill*(allM**,** agg.allRequests())**;** }  
  
 private static void rebuild(DefaultTableModel m**,** List<Object[]> rows) {  
 m.setRowCount(0)**;** rows.forEach(m::addRow)**;** }  
 private static void fill(DefaultListModel<String> m**,** List<Request> rs) {  
 m.clear()**;** for (int i = rs.size()-1**;** i >= 0**;** i--) {  
 Request r = rs.get(i)**;** m.addElement("#" + r.getId() + " " + r.getType())**;** }  
 }  
}

SystemMonitorGUI.java

package gui**;**import concurrency.engine.ConcurrencyEngine**;**import core.decorators.EncryptionDocumentDecorator**;**import core.decorators.LoggingDocumentDecorator**;**import core.documents.TextDocument**;**import core.memento.DocumentHistoryLogger**;**import javax.swing.\***;**import java.awt.\***;**import java.awt.event.WindowAdapter**;**import java.awt.event.WindowEvent**;**public class SystemMonitorGUI extends JFrame {  
  
 private final ConcurrencyEngine engine**;** public SystemMonitorGUI(int c**,** int w**,** int cap**,** int total) {  
  
 var baseDoc = new TextDocument()**;** var encDoc = new EncryptionDocumentDecorator(baseDoc)**;** var logDoc = new LoggingDocumentDecorator(encDoc)**;** var history = new DocumentHistoryLogger()**;** engine = new ConcurrencyEngine(c**,** w**,** cap**,** total**,** logDoc**,** history)**;** setTitle("Concurrency Monitor")**;** setLayout(new BorderLayout())**;** JTabbedPane tabs = new JTabbedPane()**;** tabs.add("Queue"**,** new QueueDashboard(engine.stats()))**;** tabs.add("Logs"**,** new LogsPanel())**;** add(tabs**,** BorderLayout.*CENTER*)**;** setSize(750**,** 380)**;** setLocationRelativeTo(null)**;** setDefaultCloseOperation(*EXIT\_ON\_CLOSE*)**;** addWindowListener(new WindowAdapter() {  
 @Override public void windowClosing(WindowEvent e) { engine.shutdown()**;** }  
 })**;** }  
}

Main.java

package main**;**import gui.SystemMonitorGUI**;**import javax.swing.\***;**public class Main {  
  
 public static void main(String[] args) {  
  
 SwingUtilities.*invokeLater*(() -> {  
 int clients = *ask*("Clients"**,** 5)**;** int workers = *ask*("Workers"**,** 3)**;** int capacity = *ask*("Queue capacity"**,** 128)**;** int limit = *ask*("Total requests"**,** 512)**;** new SystemMonitorGUI(clients**,** workers**,** capacity**,** limit)  
 .setVisible(true)**;** })**;** }  
  
 private static int ask(String msg**,** int def) {  
 String s = JOptionPane.*showInputDialog*(null**,** msg**,** def)**;** try { return Integer.*parseInt*(s)**;** } catch (Exception e) { return def**;** }  
 }  
}

**Приложение 2**

**Результаты тестирования**







