Coverage for **order_processing.py**: 97%

164 statements | 159 run | 5 missing | 0 excluded


```python
import csv
import time
from abc import ABC, abstractmethod
from typing import List, Any, Optional, NamedTuple
from enum import Enum


class OrderType(Enum):
    EXPORT = 'A'
    API = 'B'
    SIMPLE = 'C'
    UNKNOWN = 'UNKNOWN'


class OrderStatus(Enum):
    NEW = 'new'
    EXPORTED = 'exported'
    PROCESSED = 'processed'
    PENDING = 'pending'
    COMPLETED = 'completed'
    IN_PROGRESS = 'in_progress'
    ERROR = 'error'


class OrderError(Enum):
    EXPORT_FAILED = 'export_failed'
    API_ERROR = 'api_error'
    API_FAILURE = 'api_failure'
    DB_ERROR = 'db_error'
    UNKNOWN_TYPE = 'unknown_type'


class OrderPriority(Enum):
    LOW = 'low'
    HIGH = 'high'


class Configuration:
    HIGH_VALUE_ORDER_THRESHOLD = 150.0
    API_DATA_THRESHOLD = 50.0
    API_AMOUNT_THRESHOLD = 100.0


class ProcessingResult(NamedTuple):
    status: Optional[OrderStatus] = None
    error: Optional[OrderError] = None

    @property
    def is_success(self) -> bool:
        return self.status is not None and self.error is None


class Order:
    def __init__(self, id: int, type: OrderType, amount: float, flag: bool):
        self.id = id
        self.type = type
        self.amount = amount
        self.flag = flag
        self.status: OrderStatus = OrderStatus.NEW
        self.priority: OrderPriority = OrderPriority.LOW


class APIException(Exception):
    pass


class DatabaseException(Exception):
    pass


class FileExportException(Exception):
    pass


class APIResponse:
    def __init__(self, status: str, data: Any):
        self.status = status
        self.data = data


class DatabaseService(ABC):
    @abstractmethod
    def get_orders_by_user(self, user_id: int) -> List[Order]:
        pass

    @abstractmethod
    def update_order_status(self, order_id: int, status: OrderStatus, priority: OrderPriority) -> bool:
        pass


class APIClient(ABC):
    @abstractmethod
    def call_api(self, order_id: int) -> APIResponse:
        pass


class FileExporter(ABC):
    @abstractmethod
    def export_order_to_file(self, order: Order, user_id: int) -> None:
        pass


class CSVFileExporter(FileExporter):
    def export_order_to_file(self, order: Order, user_id: int) -> None:
        timestamp = int(time.time())
        csv_file = f'orders_type_A_{user_id}_{timestamp}.csv'
        try:
            with open(csv_file, 'w', newline='') as file_handle:
                writer = csv.writer(file_handle)
                writer.writerow(['ID', 'Type', 'Amount', 'Flag', 'Status', 'Priority'])
                writer.writerow([
                    order.id,
                    order.type.value,
                    order.amount,
                    str(order.flag).lower(),
                    order.status.value,
                    order.priority.value
                ])
                if order.amount > Configuration.HIGH_VALUE_ORDER_THRESHOLD:
                    writer.writerow(['', '', '', '', 'Note', 'High value order'])
        except IOError as e:
            raise FileExportException(f"Can not export csv: {str(e)}")
```

```python

class OrderProcessor(ABC):
    @abstractmethod
    def process(self, order: Order) -> ProcessingResult:
        pass


class ExportOrderProcessor(OrderProcessor):
    def __init__(self, file_exporter: FileExporter, user_id: int):
        self.file_exporter = file_exporter
        self.user_id = user_id

    def process(self, order: Order) -> ProcessingResult:
        try:
            order.status = OrderStatus.EXPORTED
            self.file_exporter.export_order_to_file(order, self.user_id)
            return ProcessingResult(status=OrderStatus.EXPORTED)
        except FileExportException:
            return ProcessingResult(error=OrderError.EXPORT_FAILED)


class APIOrderProcessor(OrderProcessor):
    def __init__(self, api_client: APIClient):
        self.api_client = api_client

    def determine_status(self, api_response: APIResponse, order: Order) -> ProcessingResult:
        if api_response.status != 'success':
            return ProcessingResult(error=OrderError.API_ERROR)
        if api_response.data >= Configuration.API_DATA_THRESHOLD and order.amount < Configuration.API_AMOUNT_THRESHOLD:
            return ProcessingResult(status=OrderStatus.PROCESSED)
        elif api_response.data < Configuration.API_DATA_THRESHOLD or order.flag:
            return ProcessingResult(status=OrderStatus.PENDING)
        return ProcessingResult(status=OrderStatus.ERROR)

    def process(self, order: Order) -> ProcessingResult:
        try:
            api_response = self.api_client.call_api(order.id)
            return self.determine_status(api_response, order)
        except APIException:
            return ProcessingResult(error=OrderError.API_FAILURE)


class SimpleOrderProcessor(OrderProcessor):
    def process(self, order: Order) -> ProcessingResult:
        status = OrderStatus.COMPLETED if order.flag else OrderStatus.IN_PROGRESS
        return ProcessingResult(status=status)


class UnknownOrderProcessor(OrderProcessor):
    def process(self, order: Order) -> ProcessingResult:
        return ProcessingResult(error=OrderError.UNKNOWN_TYPE)


class PriorityCalculator:
    @staticmethod
    def determine_priority(amount: float) -> OrderPriority:
        return OrderPriority.HIGH if amount > Configuration.HIGH_PRIORITY_THRESHOLD else OrderPriority.LOW


class OrderProcessingService:
    def __init__(self, db_service: DatabaseService, api_client: APIClient, file_exporter: FileExporter):
        self.db_service = db_service
        self.api_client = api_client
        self.file_exporter = file_exporter
        self.priority_calculator = PriorityCalculator()
        self.processors = {
            OrderType.EXPORT: ExportOrderProcessor,
            OrderType.API: APIOrderProcessor,
            OrderType.SIMPLE: SimpleOrderProcessor
        }

    def _get_processor(self, order: Order, user_id: int) -> OrderProcessor:
        processor_class = self.processors.get(order.type, UnknownOrderProcessor)
        if order.type == OrderType.EXPORT:
            return processor_class(self.file_exporter, user_id)
        elif order.type == OrderType.API:
            return processor_class(self.api_client)
        return processor_class()

    def process_orders(self, user_id: int) -> bool:
        try:
            orders = self.db_service.get_orders_by_user(user_id)
            if not orders:
                return False

            success = True
            for order in orders:
                processor = self._get_processor(order, user_id)
                result = processor.process(order)

                if result.is_success:
                    order.status = result.status
                else:
                    order.status = result.error

                order.priority = self.priority_calculator.determine_priority(order.amount)

                try:
                    if not self.db_service.update_order_status(order.id, order.status, order.priority):
                        order.status = OrderError.DB_ERROR
                        success = False
                except DatabaseException:
                    order.status = OrderError.DB_ERROR
                    success = False

            return success
        except Exception:
            return False
```