Extracted Code Report

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/conftest.py

```
import pytest

@pytest.fixture
def director():
    return Director()  # Replace with the actual object creation logic
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/map_file_structure.py

```
import os
import time
import argparse
import pwd
import grp
import hashlib
import mimetypes
import subprocess
import stat
from pathlib import Path

from stream_to_console import stc



def get_file_details(file_path):
    """
    Retrieves comprehensive details about a file.

    Args:
    file_path (str): Path to the file.

    Returns:
    dict: A dictionary containing various file details.
    """
    try:
        # Basic file stats
        stats = os.stat(file_path)
        file_info = {
            "size": stats.st_size,
            "last_modified": time.ctime(stats.st_mtime),
            "last_accessed": time.ctime(stats.st_atime),
            "created": time.ctime(stats.st_ctime)
        }

        # Owner and permissions
        file_info["owner"] = stat.filemode(stats.st_mode)
        file_info["uid"] = stats.st_uid
        file_info["gid"] = stats.st_gid
        # File hash for integrity
        hasher = hashlib.sha256()
        with open(file_path, 'rb') as file:
            buf = file.read()
            hasher.update(buf)
        file_info["hash"] = hasher.hexdigest()

        # Additional details based on file type
        if file_path.endswith('.py'):  # Example for Python files
            with open(file_path, 'r') as file:
                lines = file.readlines()
                file_info["line_count"] = len(lines)
                # Additional Python-specific analysis can be done here

        return file_info

    except Exception as e:
        return {"error": str(e)}


def parse_arguments():
    parser = argparse.ArgumentParser(description='Generate file structure with optional verbosity and deep s
    parser.add_argument('-v', '--verbose', action='store_true', help='Enable verbose output')
    parser.add_argument('-d', '--deep_scan', action='store_true', help='Enable deep scanning for additional fil
    return parser.parse_args()

def print_verbose(message, status):
    if status == "info":
        # Green for file details, blue for summary headers
        parts = message.split(":")
        colored_message = "\033[32m" + parts[0] + "\033[0m" if len(parts) == 1 else "\033[34m" + parts[0] + ":\0
        print(colored_message)
    else:
        # Default colors for other statuses
        color_code = "32" if status == "success" else "31"
        print(f"\033[{color_code}m{message}\033[0m")

def print_verbose_info(message):
    # Blue color for variable content
    print(f"\033[34m{message}\033[0m", end="")

def print_verbose_label(message):
    # Grey color for non-variable text
    print(f"\033[90m{message}\033[0m", end="")

def color_text(text, color_code):
    return f"\033[{color_code}m{text}\033[0m"

def format_file_size(size):
    for unit in ['B', 'KB', 'MB', 'GB', 'TB']:
        if size < 1024:
            return f"{size:.2f}{unit}"
        size /= 1024

def get_file_permissions(file_path):
    permissions = oct(os.stat(file_path).st_mode)[-3:]
    return permissions

def get_file_owner(file_path):
    uid = os.stat(file_path).st_uid
    gid = os.stat(file_path).st_gid
    user = pwd.getpwuid(uid).pw_name
    group = grp.getgrgid(gid).gr_name
    return user, group

def get_file_hash(file_path):
    sha256_hash = hashlib.sha256()
    with open(file_path, "rb") as f:
        for byte_block in iter(lambda: f.read(4096), b""):
            sha256_hash.update(byte_block)
    return sha256_hash.hexdigest()

def get_file_type(file_path):
    if os.path.islink(file_path):
        return "Symbolic Link"
    elif os.path.isdir(file_path):
        return "Directory"
    elif os.path.isfile(file_path):
        return "File"
    else:
        return "Other"

def get_mime_type(file_path):
    mime_type, _ = mimetypes.guess_type(file_path)
    return mime_type if mime_type else "Unknown"

def get_git_commit_history(file_path, script_dir):
    try:
        cmd = f"git log -n 3 --pretty=format:'%h - %s (%cr)' -- {file_path}"
        process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, cwd
        stdout, stderr = process.communicate()
        return stdout.decode().strip() if stdout else "Not available"
    except Exception as e:
        return str(e)

def print_summary(total_files, file_types, mime_types):
    print_verbose("File summary:", "info")

    # Print the summary details directly from the variables, not from the file
    print_verbose(f"Total files processed: \033[34m{total_files}\033[32m", "info")
    print_verbose("File types distribution:", "info")
    for f_type, count in file_types.items():
        print_verbose(f"  \033[32m{f_type}: \033[34m{count}\033[32m", "info")
    print_verbose("MIME types distribution:", "info")
    for m_type, count in mime_types.items():
        print_verbose(f"  \033[32m{m_type}: \033[34m{count}\033[32m", "info")
    print('')

def print_deep_scan_summary(deep_scan_details):
    if deep_scan_details:  # Check if there are any deep scan details
        print_verbose("Deep scan details:", "info")
        for file_path, details in deep_scan_details.items():
            detail_parts = [color_text(f"File: {os.path.basename(file_path)}", 34)]
            for key, value in details.items():
                detail_parts.append(color_text(f"{key.capitalize()}: {value}", 34))
            print_verbose(", ".join(detail_parts), "info")
        print('')


def generate_file_structure(script_dir, run_name, base_output_dir='file_tree/runs',
                            skip_dirs=None, include_hidden=False, deep_scan=False, verbose=False):
    if skip_dirs is None:
        skip_dirs = ['bin', 'lib', 'include', 'your_lib_folder', 'archive', '.git', '__pycache__']

    output_dir = os.path.join(base_output_dir, run_name)
    if not os.path.exists(output_dir):
        os.makedirs(output_dir)

    output_file = os.path.join(output_dir, 'file_structure.txt')
    summary_file = os.path.join(output_dir, 'summary.txt')
    error_log_file = os.path.join(output_dir, 'error_log.txt')

    total_files = 0
    file_types = {}
    mime_types = {}
    deep_scan_details = {}

    def update_distributions(file_type, mime_type):
        nonlocal total_files, file_types, mime_types
        total_files += 1
    file_types[file_type] = file_types.get(file_type, 0) + 1
    mime_types[mime_type] = mime_types.get(mime_type, 0) + 1

with open(output_file, 'w') as file_out, open(summary_file, 'w') as summary_out, open(error_log_file, 'w') a
    for root, dirs, files in os.walk(script_dir):
        if not include_hidden:
            dirs[:] = [d for d in dirs if not d.startswith('.')]
            files = [f for f in files if not f.startswith('.')]
        dirs[:] = [d for d in dirs if d not in skip_dirs]

        for f in files:
            file_path = os.path.join(root, f)
            try:
                file_stat = os.stat(file_path)
                file_size = format_file_size(file_stat.st_size)
                mod_time = time.strftime("%Y-%m-%d %H:%M:%S", time.localtime(file_stat.st_mtime))
                file_type = get_file_type(file_path)
                mime_type = get_mime_type(file_path)
                update_distributions(file_type, mime_type)

                if deep_scan:
                    file_hash = get_file_hash(file_path) if deep_scan else "N/A"
                    git_history = get_git_commit_history(file_path, script_dir) if deep_scan else "N/A"
                    deep_scan_details[file_path] = {'hash': file_hash, 'git_history': git_history}

                file_detail = f'{os.path.join(root.replace(script_dir, ""), f)} - Type: {file_type} MIME: {mime_type} S
                file_out.write(f'{file_detail} | {deep_scan_details} Modified: {mod_time}\n')

                if verbose:
                    print_verbose_label("Name: ")
                    print_verbose_info(f"{os.path.basename(file_path)}, ")
                    print_verbose_label("Type: ")
                    print_verbose_info(f"{file_type} ")
                    print_verbose_label("MIME: ")
                    print_verbose_info(f"{mime_type} ")
                    print_verbose_label("Size: ")
                    print_verbose_info(f"{file_size} ")
                    print_verbose_label("Last Modified: ")
                    print_verbose_info(f"{mod_time} ")
                    if deep_scan:
                        for file_path, details in deep_scan_details.items():
                            detail_parts = [color_text(f"File: {os.path.basename(file_path)}", 34)]
                            for key, value in details.items():
                                detail_parts.append(color_text(f"{key.capitalize()}: {value}", 34))
                            print_verbose(", ".join(detail_parts), "info")
                    print('')
            except Exception as e:
                error_log.write(f"Error processing file {file_path}: {e}\n")
                if verbose:
                    print_verbose(f"\rError processing file {file_path}: {e}", "error")

        summary_out.write(f'Total files processed: {total_files}\n')
        summary_out.write('File types distribution:\n')
        for f_type, count in file_types.items():
            summary_out.write(f'  {f_type}: {count}\n')
        summary_out.write('MIME types distribution:\n')
        for m_type, count in mime_types.items():
            summary_out.write(f'  {m_type}: {count}\n')



        if deep_scan:
            print_verbose("Deep scan details:", "info")
            for file, details in deep_scan_details.items():
                print_verbose(f"\033[32mFile: \033[34m{file}\033[32m Hash: \033[34m{details['hash']}\033[32m Gi

        if verbose:
            print_verbose(f"\rFile structure generation complete. Total files processed: {total_files}", "success")
            print_summary(total_files, file_types, mime_types)

        if verbose and deep_scan:
            print_deep_scan_summary(deep_scan_details)

        # At the end, just print the total files processed
        print(f"File map complete. Total files processed: {total_files}")

if __name__ == '__main__':
    args = parse_arguments()
    verbose = args.verbose
    deep_scan = args.deep_scan
    script_directory = os.getcwd()
    current_time = time.strftime("%Y%m%d_%H%M%S")
    run_name = f'run_{current_time}'
    generate_file_structure(script_directory, run_name, deep_scan=deep_scan, verbose=verbose)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/test_design_patterns.py

```
# # test_design_patterns.py
# New way of testing

import importlib
import sys

def run_all_tests():
    patterns = [
        "factory", "builder", "prototype", "singleton", "adapter",
        "bridge", "composite", "decorator", "facade", "flyweight",
        "proxy", "chain_of_responsibility", "command", "iterator",
        "observer", "memento", "mediator", "memoize", "state",
        "strategy", "template", "visitor"
        # Add more patterns as needed
    ]

    for pattern in patterns:
        try:
            test_module = importlib.import_module(f"{pattern}.test_{pattern}")
            print(f"\nTesting {pattern.capitalize()} Pattern:")
            test_module.run_tests()
        except ModuleNotFoundError:
            print(f"Test module for {pattern} not found.", file=sys.stderr)
        except AttributeError:
            print(f"No run_tests() function in test_{pattern}.", file=sys.stderr)

if __name__ == "__main__":
    run_all_tests()



# Old way of testing with unittest
# # Test standard design patterns
# from factory.test_factory import main as test_factory_main
# from builder.test_builder import main as test_builder_main
# from prototype.test_prototype import main as test_prototype_main
# from singleton.test_singleton import main as test_singleton_main
# from adapter.test_adapter import main as test_adapter_main
# from bridge.test_bridge import main as test_bridge_main
# from composite.test_composite import main as test_composite_main
# from decorator.test_decorator import main as test_decorator_main
# from facade.test_facade import main as test_facade_main
# from flyweight.test_flyweight import main as test_flyweight_main
# from proxy.test_proxy import main as test_proxy_main
# from chain_of_responsibility.test_chain_of_responsibility import main as test_chain_of_responsibility_mai
# from command.test_command import main as test_command_main
# from iterator.test_iterator import main as test_iterator_main
# from observer.test_observer import main as test_observer_main
# from memento.test_memento import main as test_memento_main
# from mediator.test_mediator import main as test_mediator_main
# from memoize.test_memoize import main as test_memoize_main
# from state.test_state import main as test_state_main
# from strategy.test_strategy import main as test_strategy_main
# from template.test_template import main as test_template_main
# from visitor.test_visitor import main as test_visitor_main
# # Test composite design patterns
# # from event_queue.test_queue import main as test_event_queue_main
# # from thread_pool.test_thread_pool import main as test_thread_pool_main
# # from web_scraper.test_scraper import main as test_scraper_main
# # from zip_file.test_zip_file import main as test_zip_file_main

# def run_all_tests():
#     print("Testing Factory Pattern:")
#     test_factory_main()

#     print("\n\nTesting Builder Pattern:")
#     test_builder_main()

#     print("\n\nTesting Prototype Pattern:")
#     test_prototype_main()

#     print("\n\nTesting Singleton Pattern:")
#     test_singleton_main()

#     print("\n\nTesting Adapter Pattern:")
#     test_adapter_main()

#     print("\n\nTesting Bridge Pattern:")
#     test_bridge_main()

#     print("\n\nTesting Composite Pattern:")
#     test_composite_main()

#     print("\n\nTesting Decorator Pattern:")
#     test_decorator_main()

#     print("\n\nTesting Facade Pattern:")
#     test_facade_main()

#     print("\n\nTesting Flyweight Pattern:")
#     test_flyweight_main()

#     print("\n\nTesting Proxy Pattern:")
#     test_proxy_main()

#     print("\n\nTesting Chain of Responsibility Pattern:")
#     test_chain_of_responsibility_main()

#     print("\n\nTesting Command Pattern:")
#     test_command_main()
#     print("\n\nTesting Iterator Pattern:")
#     test_iterator_main()

#     print("\n\nTesting Observer Pattern:")
#     test_observer_main()

#     print("\n\nTesting State Pattern:")
#     test_state_main()

#     print("\n\nTesting Strategy Pattern:")
#     test_strategy_main()

#     print("\n\nTesting Template Method Pattern:")
#     test_template_main()

#     print("\n\nTesting Visitor Pattern:")
#     test_visitor_main()

#     print("\n\nTesting Memento Pattern:")
#     test_memento_main()

#     print("\n\nTesting Mediator Pattern:")
#     test_mediator_main()

#     print("\n\nTesting Memoize Pattern:")
#     test_memoize_main()

#     # print("\n\nTesting Event Queue Pattern:")
#     # test_event_queue_main()

#     # print("\n\nTesting Thread Pool Pattern:")
#     # test_thread_pool_main()

#     # print("\n\nTesting Web Scraper Pattern:")
#     # test_scraper_main()

#     # print("\n\nTesting Zip File Pattern:")
#     # test_zip_file_main()

#     #================================================================================#
#     print()

# if __name__ == "__main__":
#     run_all_tests()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/copy_codebase_to_file.py

```
import os
import argparse
import logging
from datetime import datetime
import zipfile
import json
import xml.etree.ElementTree as ET
from reportlab.lib.pagesizes import letter
from reportlab.pdfgen import canvas

def write_to_pdf(file_data, output_file):
    c = canvas.Canvas(output_file, pagesize=letter)
    width, height = letter
    c.drawString(30, height - 30, "Extracted Code Report")

    y_position = height - 50
    for file_path, content in file_data.items():
        c.drawString(30, y_position, file_path)
        y_position -= 20
        for line in content.split('\n'):
            c.drawString(40, y_position, line)
            y_position -= 15
            if y_position < 40:
                c.showPage()
                y_position = height - 50

    c.save()

def ensure_directory_exists(directory):
    if not os.path.exists(directory):
        os.makedirs(directory)

def write_to_json(file_data, output_file):
    ensure_directory_exists(os.path.dirname(output_file))
    with open(output_file, 'w') as json_file:
        json.dump(file_data, json_file, indent=4)

def write_to_xml(file_data, output_file):
    ensure_directory_exists(os.path.dirname(output_file))
    root = ET.Element("files")
    for file_path, content in file_data.items():
        file_elem = ET.SubElement(root, "file", path=file_path)
        content_elem = ET.SubElement(file_elem, "content")
        content_elem.text = content

    tree = ET.ElementTree(root)
    tree.write(output_file)
```

```
def setup_logging(log_level, log_file=None):
    log_format = '%(asctime)s - %(levelname)s - %(message)s'
    logging.basicConfig(filename=log_file if log_file else None,
                        level=log_level, format=log_format)

def parse_arguments():
    parser = argparse.ArgumentParser(description='Extract Python code from a directory into separate files in
    parser.add_argument('--directory', type=str, help='Directory to scan for Python files (relative or absolute).')
    parser.add_argument('--output_folder', type=str, help='Folder to write extracted code into separate files (re
    parser.add_argument('--log', type=str, help='Optional log file')
    parser.add_argument('--log_level', type=str, default='INFO', choices=['DEBUG', 'INFO', 'WARNING', 'ERR
                        help='Set the logging level (default: INFO)')
    parser.add_argument('--min_size', type=int, default=0, help='Minimum file size in bytes')
    parser.add_argument('--max_size', type=int, default=None, help='Maximum file size in bytes')
    parser.add_argument('--before_date', type=str, default=None, help='Filter files modified before this date (Y
    parser.add_argument('--format', type=str, default='txt', choices=['txt', 'json', 'xml', 'pdf'],
                        help='Output format: txt, json, xml, or pdf (default: txt)')
    return parser.parse_args()

def is_python_file(file_path):
    return file_path.endswith('.py')

def filter_files(file_path, min_size, max_size, before_date):
    try:
        file_stat = os.stat(file_path)
        file_size = file_stat.st_size
        file_mod_time = datetime.fromtimestamp(file_stat.st_mtime)

        if (min_size is not None and file_size < min_size) or \
           (max_size is not None and file_size > max_size) or \
           (before_date is not None and file_mod_time > before_date):
            return False
        return True
    except Exception as e:
        logging.error(f"Error filtering file {file_path}: {e}")
        return False

def recursive_traverse_directory(directory, min_size, max_size, before_date):
    for root, dirs, files in os.walk(directory):
        for file in files:
            file_path = os.path.join(root, file)
            if is_python_file(file_path) and filter_files(file_path, min_size, max_size, before_date):
                yield file_path

def read_file(file_path):
    try:
        with open(file_path, 'r') as infile:
            content = infile.read()
            return content
    except IOError as e:
        logging.error(f"Error reading file {file_path}: {e}")
        return None

def write_to_single_file(file_path, content, outfile):
    outfile.write(f"\n\n# File: {file_path}\n\n")
    outfile.write(content)

def extract_python_code(directory, output_file, min_size, max_size, before_date):
    try:
        with open(output_file, 'w') as outfile:
            for file_path in recursive_traverse_directory(directory, min_size, max_size, before_date):
                content = read_file(file_path)
                if content:
                    write_to_single_file(file_path, content, outfile)
    except Exception as e:
        logging.error(f"Error while writing to file {output_file}: {e}")

def convert_date_string(date_str):
    return datetime.strptime(date_str, '%Y-%m-%d') if date_str else None

def zip_folder(output_folder, zip_file_name):
    try:
        with zipfile.ZipFile(zip_file_name, 'w', zipfile.ZIP_DEFLATED) as zipf:
            for root, dirs, files in os.walk(output_folder):
                for file in files:
                    file_path = os.path.join(root, file)
                    zipf.write(file_path, os.path.relpath(file_path, output_folder))
        logging.info(f"Folder zipped into: {zip_file_name}")
    except Exception as e:
        logging.error(f"Error zipping folder {output_folder}: {e}")

if __name__ == '__main__':
    print("Running script...")
    args = parse_arguments()
    current_dir = os.getcwd()

    # Set default for directory
    args.directory = args.directory or current_dir

    # Set default for output folder
    cwd_name = os.path.basename(current_dir)
    timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
    args.output_folder = args.output_folder or os.path.join(current_dir, f"{cwd_name}_codebase_copies", f"co
    # Ensure the output folder exists
    ensure_directory_exists(args.output_folder)

    # Generate output filename based on chosen format
    args.format = args.format or 'txt'
    output_file = os.path.join(args.output_folder, f"extracted_code_{timestamp}.{args.format}")

    setup_logging(args.log_level, args.log)

    # Convert date string to datetime object
    before_date = convert_date_string(args.before_date) if args.before_date else None

    # Process files based on the chosen format
    if args.format in ['json', 'xml', 'pdf']:
        file_data = {}
        for file_path in recursive_traverse_directory(args.directory, args.min_size, args.max_size, before_date)
            content = read_file(file_path)
            if content:
                file_data[file_path] = content

        if args.format == 'json':
            write_to_json(file_data, output_file)
        elif args.format == 'xml':
            write_to_xml(file_data, output_file)
        elif args.format == 'pdf':
            write_to_pdf(file_data, output_file)
    else:
        # Default to text format
        extract_python_code(args.directory, output_file, args.min_size, args.max_size, before_date)

    print("Script execution completed.")
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/stream_to_console.py

```
# NovaSystem/src/utils/stream_to_console.py

import traceback
import sys
import time
import art
import random
from colorama import Fore, Back, Style, init
# from ..utils.border_maker import border_maker

# ANSI Escape Codes for additional styles
ANSI_STYLES = {
    "underline": "\033[4m",
    "double_underline": "\033[21m",
    "invert_colors": "\033[7m",
    "italic": "\033[3m",
    "strikethrough": "\033[9m",
    "reset": "\033[0m"
}

# Initialize colorama
init(autoreset=True)

def apply_color(text, foreground_color=None, background_color=None, style=None):
    """Applies color and style to text."""
    colored_text = text
    if foreground_color:
        if hasattr(Fore, foreground_color.upper()):
            colored_text = getattr(Fore, foreground_color.upper()) + colored_text
        else:
            raise ValueError(f"Invalid foreground color: {foreground_color}")

    if background_color:
        if hasattr(Back, background_color.upper()):
            colored_text = getattr(Back, background_color.upper()) + colored_text
        else:
            raise ValueError(f"Invalid background color: {background_color}")

    if style:
        colored_text = style + colored_text
    return colored_text

font_options = [
    "block", "caligraphy","graffiti", "colossal",
    "sub-zero", "slant", "fancy1", "fancy2", "fancy3",
    "fancy4", "fancy5", "fancy6", "fancy7", "fancy8", "fancy9",
    "fancy10", "fancy11", "fancy12", "fancy13", "fancy14",
    "fancy15", "fancy16", "fancy17", "fancy18", "fancy19",
    "fancy20", "banner", "big", "bubble", "digital", "ivrit",
    "mirror", "script", "shadow", "speed", "stampatello",
    "term", "avatar", "barbwire", "bear", "bell", "benjamin",
    "bigchief", "binary", "broadway", "bubblebath", "bulbhead",
    "chunky", "coinstak", "contessa", "contrast", "cosmic",
    "cosmike", "cricket", "cyberlarge", "cybermedium", "cybersmall",
    "decimal", "diamond", "dietcola", "digital", "doh",
    "doom", "dotmatrix", "double", "drpepper", "eftichess",
    "eftifont", "eftipiti", "eftirobot", "eftitalic", "eftiwall",
    "eftiwater", "epic", "fender", "fourtops", "fraktur",
    "goofy", "gothic", "graceful", "gradient", "helv",
    "hollywood", "invita", "isometric1", "isometric2", "isometric3",
    "isometric4", "italic", "jazmine", "jerusalem", "katakana",
    "kban", "keyboard", "knob", "larry3d", "lcd",
    "lean", "letters", "linux", "lockergnome", "madrid",
    "marquee", "maxfour", "mike", "mini", "mirror",
    "mnemonic", "morse", "moscow", "mshebrew210", "nancyj",
    "nancyj-fancy", "nancyj-underlined", "nipples", "ntgreek", "nvscript",
    "o8", "ogre", "pawp", "peaks", "pebbles",
    "pepper", "poison", "puffy", "pyramid", "rectangles",
    "relief", "relief2", "rev", "roman", "rot13",
    "rounded", "rowancap", "rozzo", "runic", "runyc",
    "sblood", "script", "serifcap", "shadow", "short",
    "slscript", "small", "smisome1", "smkeyboard", "smscript",
    "smshadow", "smslant", "smtengwar", "speed", "stampatello",
    "standard", "starwars", "stellar", "stop", "straight",
    "tanja", "tengwar", "term", "thick", "thin",
    "threepoint", "ticks", "ticksslant", "tinker-toy", "tombstone",
    "trek", "tsalagi", "twopoint", "univers", "usaflag",
    "wavy", "weird"
]

def apply_colorama_style(bold=False, underline=False, invert_colors=False, double_underline=False, hidde
    """Returns the combined style string based on flags."""
    style_str = ''
    if bold:
        style_str += Style.BRIGHT
    if hidden:
        style_str += Style.DIM
    if underline:
        style_str += ANSI_STYLES["underline"]
    if double_underline:
        style_str += ANSI_STYLES["double_underline"]
    if invert_colors:
        style_str += ANSI_STYLES["invert_colors"]
    if italic:
        style_str += ANSI_STYLES["italic"]
    if strikethrough:
        style_str += ANSI_STYLES["strikethrough"]
    if fg_style:
        if fg_style == "DIM":
            style_str += Style.DIM
        if fg_style == "BRIGHT":
            style_str += Style.BRIGHT
        if fg_style == "NORMAL":
            style_str += Style.NORMAL
        if fg_style == "RESET_ALL":
            style_str += Style.RESET_ALL
    if bg_style:
        if bg_style == "DIM":
            style_str += Style.DIM
        if bg_style == "BRIGHT":
            style_str += Style.BRIGHT
        if bg_style == "NORMAL":
            style_str += Style.NORMAL
        if bg_style == "RESET_ALL":
            style_str += Style.RESET_ALL
    if style:
        if style == "DIM":
            style_str += Style.DIM
        if style == "BRIGHT":
            style_str += Style.BRIGHT
        if style == "NORMAL":
            style_str += Style.NORMAL
        if style == "RESET_ALL":
            style_str += Style.RESET_ALL
    return style_str

def stream_to_console(message, delay=0.0035, foreground_color=None, background_color=None, rainbow
    """
    Streams a message to the console character by character with optional delay, colors, and effects.
    """
    # Validate input types
    if not isinstance(message, str):
        raise TypeError("Message must be a string.")
    if not isinstance(delay, (float, int)):
        raise TypeError("Delay must be a number.")

    # Stream function
    try:
        # Validate delay
        delay = max(0.0001, min(delay, 1.0))  # Clamp delay

        # Style string
        style_str = apply_colorama_style(**style_flags)

        # Stream each character
        for char in message:
            if rainbow_effect:
                fg_color = random.choice(["RED", "GREEN", "YELLOW", "BLUE", "MAGENTA", "CYAN"])
                char = apply_color(char, foreground_color=fg_color, background_color=background_color, style=s
            else:
                char = apply_color(char, foreground_color, background_color, style_str)

            sys.stdout.write(char)
            sys.stdout.flush()
            time.sleep(delay)
        # Reset color at the end
        sys.stdout.write(Style.RESET_ALL)
        sys.stdout.flush()
    except Exception as e:
        exc_type, exc_value, exc_traceback = sys.exc_info()
        traceback_details = {
            'filename': exc_traceback.tb_frame.f_code.co_filename,
            'lineno': exc_traceback.tb_lineno,
            'name': exc_traceback.tb_frame.f_code.co_name,
            'type': exc_type.__name__,
            'message': str(exc_value),
        }
        error_message = "Error in stream_to_console: [{}] {}".format(traceback_details['type'], traceback_details
        error_details = "File: {}, Line: {}, In: {}".format(traceback_details['filename'], traceback_details['lineno'], tr
        sys.stderr.write(error_message + "\n" + error_details + "\n")
        sys.stderr.flush()
        raise

    print()  # Newline at the end
# Example usage and test cases remain the same

# Example usage
# stream_to_console("Hello, NovaSystem AI!", rainbow_effect=True)

# Test cases as a list of dictionaries
test_cases = [
    {"message": "Simple message with default settings."},
    {"message": "Slower text...", "delay": 0.05},
    {"message": "Red text.", "foreground_color": "red"},
    {"message": "Green text.", "foreground_color": "green"},
    {"message": "Green on blue.", "foreground_color": "green", "background_color": "blue"},
    {"message": "Rainbow effect!", "rainbow_effect": True},
    {"message": "Slower rainbow text...", "delay": 0.07, "rainbow_effect": True},
    {"message": "Green on red, slowly.", "delay": 0.05, "foreground_color": "green", "background_color": "red"
    {"message": "Bold text.", "bold": True},
    {"message": "Underlined text.", "underline": True},
    {"message": "Inverted colors.", "invert_colors": True},
    {"message": "Blue background.", "background_color": "blue"},
    {"message": "Cyan text on yellow.", "foreground_color": "cyan", "background_color": "yellow"},
    {"message": "Double underline.", "double_underline": True},
    {"message": "Hidden text.", "hidden": True},
    {"message": "Slower inverted rainbow text...", "delay": 0.07, "rainbow_effect": True, "invert_colors": True},
    {"message": "Italicized text.", "italic": True},
    {"message": "Strikethrough text.", "strikethrough": True},
]
```

```
def test():
    # Generate ASCII art with a random font
  random_font = random.choice(font_options)
  random_ascii_art = art.text2art("NovaSystem", font=random_font)

  # Stream the ASCII art first
  stream_to_console(random_ascii_art, delay=0.0004)

  # Stream each test case
  for case in test_cases:
      stream_to_console(**case)

stc = stream_to_console

if __name__ == "__main__":
    test()
```
/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/proxy/proxy.py
```
from abc import ABC, abstractmethod
import datetime
now = datetime.datetime.now()

class Subject(ABC):
    """
    The Subject interface declares common operations for both RealSubject and the Proxy.
    """
    @abstractmethod
    def request(self) -> None:
        pass

class RealSubject(Subject):
    """
    The RealSubject contains core business logic.
    """
    def request(self) -> None:
        print("RealSubject: Handling request.")

class Proxy(Subject):
    """
    The Proxy has an interface identical to the RealSubject.
    """
    def __init__(self, real_subject: RealSubject) -> None:
        self._real_subject = real_subject

    def request(self) -> None:
        if self.check_access():
            self._real_subject.request()
            self.log_access()
    def check_access(self) -> bool:
        print("Proxy: Checking access prior to firing a real request.")
        return True

    def log_access(self) -> None:
        print("Proxy: Logging the time of request.", end="")
        print(f"Time: {now.time()}")

# Client code example
def client_code(subject: Subject) -> None:
    subject.request()

# Example usage
if __name__ == "__main__":
    real_subject = RealSubject()
    proxy = Proxy(real_subject)

    print("Client: Executing with RealSubject:")
    client_code(real_subject)

    print("\nClient: Executing with Proxy:")
    client_code(proxy)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/proxy/__init__.py

```
from .proxy import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/proxy/test_proxy.py

```
from proxy import RealSubject, Proxy, client_code

def test_real_subject():
    print("Testing RealSubject:")
    real_subject = RealSubject()
    client_code(real_subject)

def test_proxy():
    print("\nTesting Proxy:")
    real_subject = RealSubject()
    proxy = Proxy(real_subject)
    client_code(proxy)

def main():
    test_real_subject()
    test_proxy()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/decorator/decorator.py

```
class AIComponent:
    """
    Base AIComponent interface defines operations that can be altered by decorators.
    """
    def operation(self) -> str:
        pass

class ConcreteAIComponent(AIComponent):
    """
    Concrete AIComponents provide default implementations of the operations.
    """
    def operation(self) -> str:
        return "ConcreteAIComponent"

class Decorator(AIComponent):
    """
    Base Decorator class follows the same interface as other components.
    """
    _component: AIComponent = None

    def __init__(self, component: AIComponent) -> None:
        self._component = component

    def operation(self) -> str:
        return self._component.operation()

class LoggingDecorator(Decorator):
    """
    Concrete Decorator that adds logging functionality.
    """
    def operation(self) -> str:
        # Additional behavior before calling the wrapped object
        result = self._component.operation()
        # Additional behavior after calling the wrapped object
        return f"LoggingDecorator({result})"

class PerformanceDecorator(Decorator):
    """
    Concrete Decorator that adds performance tracking functionality.
    """
    def operation(self) -> str:
        # Performance tracking behavior
        result = self._component.operation()
        # Additional behavior
        return f"PerformanceDecorator({result})"
```

```
# Client code
def client_code(component: AIComponent) -> None:
    print(f"RESULT: {component.operation()}", end="")

# Example usage
if __name__ == "__main__":
    simple_component = ConcreteAIComponent()
    decorated_component = PerformanceDecorator(LoggingDecorator(simple_component))

    print("Client: I've got a component with additional behaviors:")
    client_code(decorated_component)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/decorator/__init__.py

```
from .decorator import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/decorator/test_decorator.p

```
from .decorator import ConcreteAIComponent, LoggingDecorator, PerformanceDecorator

def test_decorator_pattern():
    # Testing with the basic AI component
    basic_component = ConcreteAIComponent()
    print("Basic AI Component:", basic_component.operation())

    # Adding Logging functionality
    logged_component = LoggingDecorator(basic_component)
    print("Logged AI Component:", logged_component.operation())

    # Adding Performance tracking on top of Logging
    perf_logged_component = PerformanceDecorator(logged_component)
    print("Performance Tracked and Logged AI Component:", perf_logged_component.operation())

def main():
    test_decorator_pattern()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/bridge/bridge.py

```
# bridge.py

from __future__ import annotations
from abc import ABC, abstractmethod

class Abstraction:
    """
    The Abstraction defines the interface for the 'control' part of the two
    class hierarchies. It maintains a reference to an object of the
    Implementation hierarchy and delegates all of the real work to this object.
    """
    def __init__(self, implementation: Implementation) -> None:
        self.implementation = implementation

    def operation(self) -> str:
        return (f"Abstraction: Base operation with:\n"
                f"{self.implementation.operation_implementation()}")

class ExtendedAbstraction(Abstraction):
    """
    You can extend the Abstraction without changing the Implementation classes.
    """
    def operation(self) -> str:
        return (f"ExtendedAbstraction: Extended operation with:\n"
                f"{self.implementation.operation_implementation()}")

class Implementation(ABC):
    """
    The Implementation defines the interface for all implementation classes. It
    doesn't have to match the Abstraction's interface. In fact, the two
    interfaces can be entirely different. Typically the Implementation interface
    provides only primitive operations, while the Abstraction defines higher-
    level operations based on those primitives.
    """
    @abstractmethod
    def operation_implementation(self) -> str:
        pass

class ConcreteImplementationA(Implementation):
    def operation_implementation(self) -> str:
        return "ConcreteImplementationA: Here's the result on the platform A."

class ConcreteImplementationB(Implementation):
    def operation_implementation(self) -> str:
        return "ConcreteImplementationB: Here's the result on the platform B."

def client_code(abstraction: Abstraction) -> None:
    """
    Except for the initialization phase, where an Abstraction object gets linked
    with a specific Implementation object, the client code should only depend on
    the Abstraction class. This way the client code can support any abstraction-
    implementation combination.
    """
    print(abstraction.operation(), end="")

if __name__ == "__main__":
    """
    The client code should be able to work with any pre-configured abstraction-
    implementation combination.
    """
    implementation = ConcreteImplementationA()
    abstraction = Abstraction(implementation)
    client_code(abstraction)

    print("\n")

    implementation = ConcreteImplementationB()
    abstraction = ExtendedAbstraction(implementation)
    client_code(abstraction)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/bridge/__init__.py

```
from .bridge import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/bridge/test_bridge.py

```
# test_bridge.py

from bridge import Abstraction, ExtendedAbstraction, ConcreteImplementationA, ConcreteImplementationB

def test_abstraction_with_concrete_implementation_a():
    """
    Test Abstraction with ConcreteImplementationA.
    """
    implementation = ConcreteImplementationA()
    abstraction = Abstraction(implementation)
    result = abstraction.operation()

    assert result == "Abstraction: Base operation with:\nConcreteImplementationA: Here's the result on the pl
        "Abstraction with ConcreteImplementationA failed"

    print("PASS: Abstraction with ConcreteImplementationA")

def test_abstraction_with_concrete_implementation_b():
    """
    Test Abstraction with ConcreteImplementationB.
    """
    implementation = ConcreteImplementationB()
    abstraction = Abstraction(implementation)
    result = abstraction.operation()

    assert result == "Abstraction: Base operation with:\nConcreteImplementationB: Here's the result on the pl
        "Abstraction with ConcreteImplementationB failed"

    print("PASS: Abstraction with ConcreteImplementationB")
```

```
def test_extended_abstraction_with_concrete_implementation_a():
    """
    Test ExtendedAbstraction with ConcreteImplementationA.
    """
    implementation = ConcreteImplementationA()
    abstraction = ExtendedAbstraction(implementation)
    result = abstraction.operation()

    assert result == "ExtendedAbstraction: Extended operation with:\nConcreteImplementationA: Here's the re
        "ExtendedAbstraction with ConcreteImplementationA failed"

    print("PASS: ExtendedAbstraction with ConcreteImplementationA")

def test_extended_abstraction_with_concrete_implementation_b():
    """
    Test ExtendedAbstraction with ConcreteImplementationB.
    """
    implementation = ConcreteImplementationB()
    abstraction = ExtendedAbstraction(implementation)
    result = abstraction.operation()

    assert result == "ExtendedAbstraction: Extended operation with:\nConcreteImplementationB: Here's the re
        "ExtendedAbstraction with ConcreteImplementationB failed"

    print("PASS: ExtendedAbstraction with ConcreteImplementationB")

def main():
    """
    Main function to run the Bridge pattern tests.
    """
    print("Testing Bridge Pattern Implementations:")
    test_abstraction_with_concrete_implementation_a()
    test_abstraction_with_concrete_implementation_b()
    test_extended_abstraction_with_concrete_implementation_a()
    test_extended_abstraction_with_concrete_implementation_b()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/template/__init__.py

```
from .template import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/template/template.py

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    """
    The Abstract Class defines a template method that contains a skeleton of
    some algorithm, composed of calls to (usually) abstract primitive operations.
    Concrete subclasses should implement these operations, but leave the
    template method itself intact.
    """
    def template_method(self) -> None:
        self.base_operation1()
        self.required_operations1()
        self.base_operation2()
        self.hook1()
        self.required_operations2()
        self.base_operation3()
        self.hook2()

    # These operations already have implementations.
    def base_operation1(self):
        print("AbstractClass says: I am doing the bulk of the work")

    def base_operation2(self):
        print("AbstractClass says: But I let subclasses override some operations")

    def base_operation3(self):
        print("AbstractClass says: But I am doing the majority of the work anyway")

    # These operations have to be implemented in subclasses.
    @abstractmethod
    def required_operations1(self):
        pass

    @abstractmethod
    def required_operations2(self):
        pass

    # These are "hooks." Subclasses may override them, but it's not mandatory
    # since the hooks already have default (but empty) implementation.
    def hook1(self):
        pass

    def hook2(self):
        pass

class ConcreteClass1(AbstractClass):
    def required_operations1(self):
        print("ConcreteClass1 says: Implemented Operation1")

    def required_operations2(self):
        print("ConcreteClass1 says: Implemented Operation2")
```

```
class ConcreteClass2(AbstractClass):
    def required_operations1(self):
        print("ConcreteClass2 says: Implemented Operation1")

    def required_operations2(self):
        print("ConcreteClass2 says: Implemented Operation2")

    def hook1(self):
        print("ConcreteClass2 says: Overridden Hook1")

# Example usage
if __name__ == "__main__":
    print("Same client code can work with different subclasses:")
    concrete_class1 = ConcreteClass1()
    concrete_class1.template_method()

    print("\nSame client code can work with different subclasses:")
    concrete_class2 = ConcreteClass2()
    concrete_class2.template_method()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/template/test_template.py

```
import unittest
from unittest.mock import patch
from template import ConcreteClass1, ConcreteClass2

class TestConcreteClass1(unittest.TestCase):
    @patch('sys.stdout')
    def test_required_operations1(self, mock_stdout):
        concrete_class1 = ConcreteClass1()
        concrete_class1.required_operations1()
        mock_stdout.write.assert_called_with("ConcreteClass1 says: Implemented Operation1\n")

    @patch('sys.stdout')
    def test_required_operations2(self, mock_stdout):
        concrete_class1 = ConcreteClass1()
        concrete_class1.required_operations2()
        mock_stdout.write.assert_called_with("ConcreteClass1 says: Implemented Operation2\n")

class TestConcreteClass2(unittest.TestCase):
    @patch('sys.stdout')
    def test_required_operations1(self, mock_stdout):
        concrete_class2 = ConcreteClass2()
        concrete_class2.required_operations1()
        mock_stdout.write.assert_called_with("ConcreteClass2 says: Implemented Operation1\n")

    @patch('sys.stdout')
    def test_required_operations2(self, mock_stdout):
        concrete_class2 = ConcreteClass2()
        concrete_class2.required_operations2()
        mock_stdout.write.assert_called_with("ConcreteClass2 says: Implemented Operation2\n")

    @patch('sys.stdout')
    def test_hook1(self, mock_stdout):
        concrete_class2 = ConcreteClass2()
        concrete_class2.hook1()
        mock_stdout.write.assert_called_with("ConcreteClass2 says: Overridden Hook1\n")

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/facade/facade.py

```
class AIProcessingSubsystem:
    """
    A subsystem that might perform AI-related tasks.
    """
    def initialize(self) -> str:
        return "AIProcessingSubsystem: Initialized and ready to process."

    def process_data(self, data) -> str:
        return f"AIProcessingSubsystem: Processing data - {data}"

class DataAnalysisSubsystem:
    """
    A subsystem for data analysis.
    """
    def analyze(self, data) -> str:
        return f"DataAnalysisSubsystem: Analyzing data - {data}"

class NovaSystemFacade:
    """
    The Facade class provides a simple interface to the complex logic of NovaSystem's subsystems.
    """
    def __init__(self) -> None:
        self._ai_processor = AIProcessingSubsystem()
        self._data_analyzer = DataAnalysisSubsystem()

    def process_and_analyze_data(self, data) -> str:
        results = []
        results.append(self._ai_processor.initialize())
        results.append(self._ai_processor.process_data(data))
        results.append(self._data_analyzer.analyze(data))
        return "\n".join(results)

# Client Code
def client_code(facade: NovaSystemFacade) -> None:
    print(facade.process_and_analyze_data("Sample Data"), end="")

# Example usage
if __name__ == "__main__":
    facade = NovaSystemFacade()
    client_code(facade)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/facade/__init__.py

```
from .facade import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/facade/test_facade.py

```
from facade import NovaSystemFacade, client_code

def test_novasystem_facade():
    # Creating the Facade instance
    novasystem_facade = NovaSystemFacade()

    # Simulating client interaction with the facade
    print("Testing NovaSystem Facade:")
    client_code(novasystem_facade)

def main():
    test_novasystem_facade()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/observer/observer.py

```
from abc import ABC, abstractmethod

class Observer(ABC):
    def __init__(self):
        self.is_active = True

    @abstractmethod
    def update(self, subject) -> None:
        pass

    def activate(self):
        self.is_active = True

    def deactivate(self):
        self.is_active = False
    def is_observer_active(self) -> bool:
        return self.is_active

    def handle_error(self, error: Exception):
        print(f"Observer error: {error}")

    def pre_update(self):
        pass

    def post_update(self):
        pass


# Example of a concrete observer class with expanded functionality
class AdvancedObserver(Observer):
    def __init__(self):
        super().__init__()

    def update(self, subject) -> None:
        if not self.is_active:
            return

        if subject is None:
            raise ValueError("Subject cannot be None")  # Directly raise the exception

        try:
            self.pre_update()
            # Ensure 'subject' has attribute 'state' before trying to access it
            state = getattr(subject, 'state', 'No state')  # Default value if 'state' is not present
            print(f"AdvancedObserver updated with new state: {state}")
            self.post_update()
        except Exception as e:
            self.handle_error(e)

    def pre_update(self):
        print("Preparing to update AdvancedObserver.")

    def post_update(self):
        print("AdvancedObserver update complete.")

    def handle_error(self, error: Exception):
        print(f"Error in AdvancedObserver: {error}")
        # Optionally, you can re-raise the exception if needed for tests
        raise error
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/observer/test_observer.py

```
# # DesignPatterns/observer/test_observer.py
```

```
import pytest
from io import StringIO
from unittest.mock import patch
from observer import AdvancedObserver

class MockSubject:
    """ A mock subject class for testing the observer. """
    def __init__(self):
        self.state = None

    def change_state(self, new_state):
        self.state = new_state

@pytest.fixture
def observer():
    """ Fixture to create an AdvancedObserver instance. """
    return AdvancedObserver()

@pytest.fixture
def mock_subject():
    """ Fixture to create a MockSubject instance. """
    return MockSubject()

def test_activation(observer):
    """ Test if the observer activates and deactivates correctly. """
    observer.deactivate()
    assert not observer.is_observer_active()

    observer.activate()
    assert observer.is_observer_active()

def test_update_when_active(observer, mock_subject):
    """ Test if the observer updates its state when active. """
    with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
        observer.activate()
        mock_subject.change_state("new_state")
        observer.update(mock_subject)
        assert "AdvancedObserver updated with new state: new_state" in mock_stdout.getvalue()

def test_no_update_when_inactive(observer, mock_subject):
    """ Test if the observer does not update its state when inactive. """
    with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
        observer.deactivate()
        mock_subject.change_state("new_state")
        observer.update(mock_subject)
        assert mock_stdout.getvalue() == ""
```

```
def test_error_handling(observer):
    """ Test the error handling in the observer. """
    with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
        observer.activate()
        with pytest.raises(ValueError) as exc_info:
            observer.update(None)

        assert "Subject cannot be None" == str(exc_info.value)

def test_pre_update_hook(observer, mock_subject):
    """ Test the execution of the pre-update hook. """
    with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
        observer.activate()
        observer.update(mock_subject)
        assert "Preparing to update AdvancedObserver." in mock_stdout.getvalue()

def test_post_update_hook(observer, mock_subject):
    """ Test the execution of the post-update hook. """
    with patch('sys.stdout', new_callable=StringIO) as mock_stdout:
        observer.activate()
        observer.update(mock_subject)
        assert "AdvancedObserver update complete." in mock_stdout.getvalue()


# import unittest
# from unittest.mock import patch
# from observer import Observer, AdvancedObserver

# class MockSubject:
#     """
#     A mock subject class to simulate state changes for testing observers.
#     """
#     def __init__(self):
#         self.state = None

#     def change_state(self, new_state):
#         self.state = new_state

# class TestObserver(unittest.TestCase):
#     """
#     Test suite for the Observer class and its functionalities.
#     """

#     def setUp(self):
#         self.subject = MockSubject()
#         self.observer = AdvancedObserver()
#     def test_activation(self):
#         """ Test if the observer correctly activates and deactivates. """
#         self.observer.deactivate()
#         self.assertFalse(self.observer.is_observer_active())

#         self.observer.activate()
#         self.assertTrue(self.observer.is_observer_active())

#     def test_update_when_active(self):
#         """ Test if the observer updates its state when active. """
#         with patch('sys.stdout') as mock_stdout:
#             self.subject.change_state("new_state")
#             self.observer.activate()
#             self.observer.update(self.subject)
#             self.assertIn("AdvancedObserver updated with new state: new_state", mock_stdout.getvalue())

#     def test_no_update_when_inactive(self):
#         """ Test if the observer does not update its state when inactive. """
#         with patch('sys.stdout') as mock_stdout:
#             self.subject.change_state("new_state")
#             self.observer.deactivate()
#             self.observer.update(self.subject)
#             self.assertEqual(mock_stdout.getvalue(), "")

#     def test_error_handling(self):
#         with patch('sys.stdout', new_callable=unittest.mock.StringIO) as mock_stdout:
#             self.observer.activate()
#             with self.assertRaises(ValueError) as context:
#                 self.observer.update(None)  # Passing None should trigger an error in the observer
#             self.assertEqual(str(context.exception), "Subject cannot be None")

#     def test_pre_update_hook(self):
#         """ Test the execution of the pre-update hook. """
#         with patch('sys.stdout') as mock_stdout:
#             self.observer.activate()
#             self.observer.update(self.subject)
#             self.assertIn("Preparing to update AdvancedObserver.", mock_stdout.getvalue())

#     def test_post_update_hook(self):
#         """ Test the execution of the post-update hook. """
#         with patch('sys.stdout') as mock_stdout:
#             self.observer.activate()
#             self.observer.update(self.subject)
#             self.assertIn("AdvancedObserver update complete.", mock_stdout.getvalue())

# def main():
```

```
#     unittest.main()

# if __name__ == '__main__':
#     main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/observer/__init__.py

```
from .observer import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memento/memento.py

```
# memento.py
from datetime import datetime
from typing import List

class Memento:
    """
    The Memento interface provides a way to retrieve the memento's metadata,
    such as creation date or name. It doesn't expose the Originator's state.
    """
    def get_name(self) -> str:
        pass

    def get_date(self) -> str:
        pass

class ConcreteMemento(Memento):
    def __init__(self, state: str) -> None:
        self._state = state
        self._date = str(datetime.now())[:19]

    def get_state(self) -> str:
        return self._state

    def get_name(self) -> str:
        return f"{self._date} / ({self._state[0:9]}...)"

    def get_date(self) -> str:
        return self._date

class Originator:
    """
    The Originator holds an important state that can change over time.
    It defines methods for saving and restoring the state from a Memento.
    """
    _state = None

    def __init__(self, state: str) -> None:
        self._state = state
        print(f"Originator: My initial state is: {self._state}")
    def do_something(self) -> None:
        print("Originator: I'm doing something important.")
        self._state = f"state_{datetime.now().timestamp()}"
        print(f"Originator: and my state has changed to: {self._state}")

    def save(self) -> Memento:
        return ConcreteMemento(self._state)

    def restore(self, memento: Memento) -> None:
        self._state = memento.get_state()
        print(f"Originator: My state has changed to: {self._state}")

class Caretaker:
    """
    The Caretaker works with Mementos via the base Memento interface.
    It can store and restore the Originator's state.
    """
    def __init__(self, originator: Originator) -> None:
        self._mementos = []
        self._originator = originator

    def backup(self) -> None:
        print("\nCaretaker: Saving Originator's state...")
        self._mementos.append(self._originator.save())

    def undo(self) -> None:
        if not self._mementos:
            return

        memento = self._mementos.pop()
        print(f"Caretaker: Restoring state to: {memento.get_name()}")
        self._originator.restore(memento)

    def show_history(self) -> None:
        print("Caretaker: Here's the list of mementos:")
        for memento in self._mementos:
            print(memento.get_name())

# Example usage
if __name__ == "__main__":
    originator = Originator("Initial State")
    caretaker = Caretaker(originator)

    caretaker.backup()
    originator.do_something()
    caretaker.backup()
    originator.do_something()

    caretaker.backup()
    originator.do_something()

    caretaker.show_history()

    print("\nClient: Now, let's rollback!\n")
    caretaker.undo()

    print("\nClient: Once more!\n")
    caretaker.undo()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memento/test_memento.p

```
import unittest
from unittest.mock import patch
from memento import Memento, ConcreteMemento, Originator, Caretaker

class TestMementoPattern(unittest.TestCase):
    def setUp(self):
        self.originator = Originator("Initial State")
        self.caretaker = Caretaker(self.originator)

    def test_memento_creation(self):
        """Test the creation of a memento and its properties."""
        memento = self.originator.save()
        self.assertIsInstance(memento, ConcreteMemento)
        self.assertTrue(memento.get_name().startswith("20"))  # Assuming current year
        self.assertTrue(memento.get_date().startswith("20"))  # Assuming current year

    def test_state_restoration(self):
        """Test the restoration of the state in the originator from a memento."""
        self.originator._state = "New State"
        memento = self.originator.save()
        self.originator._state = "Another State"
        self.originator.restore(memento)
        self.assertEqual(self.originator._state, "New State")

    def test_caretaker_memento_management(self):
        """Test the caretaker's ability to store and retrieve mementos."""
        self.caretaker.backup()
        self.caretaker.backup()
        self.assertEqual(len(self.caretaker._mementos), 2)

    def test_caretaker_undo_functionality(self):
        """Test the caretaker's undo functionality."""
        self.originator._state = "State A"
        self.caretaker.backup()
        self.originator._state = "State B"
        self.caretaker.backup()
        self.caretaker.undo()
        self.assertEqual(self.originator._state, "State A")

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memento/__init__.py

```
from .memento import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/adapter/adapter.py

```
# adapter.py

class Target:
    """
    The Target defines the domain-specific interface used by the client code.
    """
    def request(self) -> str:
        return "Target: The default target's behavior."


class Adaptee:
    """
    The Adaptee contains some useful behavior, but its interface is incompatible
    with the existing client code. The Adaptee needs some adaptation before the
    client code can use it.
    """
    def specific_request(self) -> str:
        return ".eetpadA eht fo roivaheb laicepS"


# Inheritance-based Adapter
class AdapterInheritance(Target, Adaptee):
    """
    The Adapter makes the Adaptee's interface compatible with the Target's
    interface via multiple inheritance.
    """
    def request(self) -> str:
        return f"Adapter (Inheritance): (TRANSLATED) {self.specific_request()[::-1]}"


# Composition-based Adapter
```

```
class AdapterComposition(Target):
    """
    The Adapter makes the Adaptee's interface compatible with the Target's
    interface via composition.
    """
    def __init__(self, adaptee: Adaptee):
        self.adaptee = adaptee

    def request(self) -> str:
        return f"Adapter (Composition): (TRANSLATED) {self.adaptee.specific_request()[::-1]}"


def client_code(target: Target):
    """
    The client code supports all classes that follow the Target interface.
    """
    print(target.request(), end="\n\n")


if __name__ == "__main__":
    print("Client: I can work just fine with the Target objects:")
    target = Target()
    client_code(target)

    adaptee = Adaptee()
    print("Client: The Adaptee class has a weird interface. See, I don't understand it:")
    print(f"Adaptee: {adaptee.specific_request()}", end="\n\n")

    print("Client: But I can work with it via the Inheritance-based Adapter:")
    adapter_inheritance = AdapterInheritance()
    client_code(adapter_inheritance)

    print("Client: And also with the Composition-based Adapter:")
    adapter_composition = AdapterComposition(adaptee)
    client_code(adapter_composition)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/adapter/__init__.py

```
from .adapter import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/adapter/test_adapter.py

```
# test_adapter.py

from adapter import Target, Adaptee, AdapterInheritance, AdapterComposition

def test_adapter_inheritance():
    """
    Test the inheritance-based Adapter.
    """
    adaptee = Adaptee()
    adapter = AdapterInheritance()

    assert adapter.request() == f"Adapter (Inheritance): (TRANSLATED) {adaptee.specific_request()[::-1]}", \
        "AdapterInheritance does not correctly adapt Adaptee"

    print("PASS: Inheritance-based Adapter test")

def test_adapter_composition():
    """
    Test the composition-based Adapter.
    """
    adaptee = Adaptee()
    adapter = AdapterComposition(adaptee)

    assert adapter.request() == f"Adapter (Composition): (TRANSLATED) {adaptee.specific_request()[::-1]}", 
        "AdapterComposition does not correctly adapt Adaptee"

    print("PASS: Composition-based Adapter test")

def main():
    """
    Main function to run the adapter tests.
    """
    print("Testing Adapter Pattern Implementations:")
    test_adapter_inheritance()
    test_adapter_composition()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/prototype/__init__.py

```
from .prototype import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/prototype/test_prototype.p

```
import copy
from prototype import NovaComponent, SelfReferencingEntity

def test_shallow_copy(nova_component):
    shallow_copied_component = copy.copy(nova_component)
    print("Testing Shallow Copy:")

    # Modifying the shallow copy and testing its effect on the original
    shallow_copied_component.some_list_of_objects.append("new item")
    if "new item" in nova_component.some_list_of_objects:
        print("Shallow copy modification reflected in the original object.")
    else:
        print("Shallow copy modification not reflected in the original object.")
```

```
def test_deep_copy(nova_component):
    deep_copied_component = copy.deepcopy(nova_component)
    print("\nTesting Deep Copy:")

    # Modifying the deep copy and testing its effect on the original
    deep_copied_component.some_list_of_objects.append("new deep item")
    if "new deep item" in nova_component.some_list_of_objects:
        print("Deep copy modification reflected in the original object.")
    else:
        print("Deep copy modification not reflected in the original object.")

def main():
    list_of_objects = [1, {1, 2, 3}, [1, 2, 3]]
    circular_ref = SelfReferencingEntity()
    nova_component = NovaComponent(23, list_of_objects, circular_ref)
    circular_ref.set_parent(nova_component)

    test_shallow_copy(nova_component)
    test_deep_copy(nova_component)

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/prototype/prototype.py

```
import copy

class SelfReferencingEntity:
    def __init__(self):
        self.parent = None

    def set_parent(self, parent):
        self.parent = parent

class NovaComponent:
    def __init__(self, some_int, some_list_of_objects, some_circular_ref):
        self.some_int = some_int
        self.some_list_of_objects = some_list_of_objects
        self.some_circular_ref = some_circular_ref

    def __copy__(self):
        some_list_of_objects = copy.copy(self.some_list_of_objects)
        some_circular_ref = copy.copy(self.some_circular_ref)

        new = self.__class__(
            self.some_int, some_list_of_objects, some_circular_ref
        )
        new.__dict__.update(self.__dict__)

        return new

    def __deepcopy__(self, memo=None):
        if memo is None:
            memo = {}

        some_list_of_objects = copy.deepcopy(self.some_list_of_objects, memo)
        some_circular_ref = copy.deepcopy(self.some_circular_ref, memo)

        new = self.__class__(
            self.some_int, some_list_of_objects, some_circular_ref
        )
        new.__dict__ = copy.deepcopy(self.__dict__, memo)

        return new

# Example usage
if __name__ == "__main__":
    list_of_objects = [1, {1, 2, 3}, [1, 2, 3]]
    circular_ref = SelfReferencingEntity()
    nova_component = NovaComponent(23, list_of_objects, circular_ref)
    circular_ref.set_parent(nova_component)

    shallow_copied_component = copy.copy(nova_component)
    deep_copied_component = copy.deepcopy(nova_component)

    # Test and demonstrate the differences between shallow and deep copy
    # ... (Similar to the provided example code)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/state/__init__.py

```
from .state import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/state/test_state.py

```
import unittest
from state import Context, ConcreteStateA, ConcreteStateB, ConcreteStateC, ConcreteStateD, StateContex

class TestStatePattern(unittest.TestCase):
    def test_initial_state(self):
        """Test the initial state setup in the context."""
        context = Context(ConcreteStateA())
        self.assertIsInstance(context.state, ConcreteStateA)

    def test_state_transition(self):
        """Test state transitions based on different conditions."""
        context = Context(ConcreteStateA())
        context.set_condition(True)  # Should transition to ConcreteStateB
        context.request()
        self.assertIsInstance(context.state, ConcreteStateB)

        context.set_condition(False)  # Should transition to ConcreteStateA
        context.request()
        self.assertIsInstance(context.state, ConcreteStateA)

    def test_special_case_handling(self):
        """Test the handling of special cases."""
        context = Context(ConcreteStateC())
        context.set_special_case(True)  # Should transition to ConcreteStateD
        context.request()
        self.assertIsInstance(context.state, ConcreteStateD)

        context.set_special_case(False)  # Should transition to ConcreteStateA
        context.request()
        self.assertIsInstance(context.state, ConcreteStateA)

    # Optional: Add a test for exception handling if relevant

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/state/state.py

```
from abc import ABC, abstractmethod
from dataclasses import dataclass
from typing import Callable, Optional

@dataclass
class StateContext:
    condition: bool = False
    special_case: bool = False

class State(ABC):
    @abstractmethod
    def handle(self, context: StateContext) -> None:
        pass

class ConcreteStateA(State):
    def handle(self, context: StateContext) -> None:
        print("State A handling context.")
        next_state = ConcreteStateB() if context.condition else ConcreteStateC()
        context.change_state(next_state)
```

```
class ConcreteStateB(State):
    def handle(self, context: StateContext) -> None:
        print("State B handling context.")
        context.change_state(ConcreteStateA())

class ConcreteStateC(State):
    def handle(self, context: StateContext) -> None:
        print("State C handling context.")
        next_state = ConcreteStateD() if context.special_case else ConcreteStateA()
        context.change_state(next_state)

class ConcreteStateD(State):
    def handle(self, context: StateContext) -> None:
        print("State D handling context (Special Case).")
        context.change_state(ConcreteStateA())

class Context:
    def __init__(self, state: State):
        self.state = state
        self.context_data = StateContext()

    def change_state(self, state: State) -> None:
        self.state = state

    def request(self) -> None:
        try:
            self.state.handle(self.context_data)
        except Exception as e:
            print(f"Error occurred: {e}")

    def set_condition(self, condition: bool) -> None:
        self.context_data.condition = condition

    def set_special_case(self, special_case: bool) -> None:
        self.context_data.special_case = special_case

# Example usage
if __name__ == "__main__":
    context = Context(ConcreteStateA())
    context.request()
    context.set_condition(True)
    context.request()
    context.set_special_case(True)
    context.request()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/composite/test_composite

```
# test_composite.py
```

```
from composite import Leaf, Composite

def test_leaf_operation():
    """
    Test the operation of a leaf component.
    """
    leaf = Leaf()
    assert leaf.operation() == "Leaf", "Leaf operation did not return expected result."
    print("PASS: Leaf operation test")

def test_composite_single_child():
    """
    Test a composite with a single child.
    """
    leaf = Leaf()
    composite = Composite()
    composite.add(leaf)

    assert composite.operation() == "Branch(Leaf)", "Composite operation with one child did not return expec
    print("PASS: Composite single child test")

def test_composite_multiple_children():
    """
    Test a composite with multiple children.
    """
    composite = Composite()
    composite.add(Leaf())
    composite.add(Leaf())

    assert composite.operation() == "Branch(Leaf+Leaf)", "Composite operation with multiple children did not
    print("PASS: Composite multiple children test")

def main():
    """
    Main function to run the composite pattern tests.
    """
    print("Testing Composite Pattern:")
    test_leaf_operation()
    test_composite_single_child()
    test_composite_multiple_children()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/composite/__init__.py

```
from .composite import *
```

```
# composite.py

from __future__ import annotations
from abc import ABC, abstractmethod
from typing import List


class Component(ABC):
    """
    The base Component class declares common operations for both simple and
    complex objects of a composition.
    """

    def __init__(self) -> None:
        self._parent: Component = None

    @property
    def parent(self) -> Component:
        return self._parent

    @parent.setter
    def parent(self, parent: Component):
        self._parent = parent

    def add(self, component: Component) -> None:
        pass

    def remove(self, component: Component) -> None:
        pass

    def is_composite(self) -> bool:
        return False

    @abstractmethod
    def operation(self) -> str:
        pass


class Leaf(Component):
    """
    The Leaf class represents the end objects of a composition. A leaf can't
    have any children. Usually, it's the Leaf objects that do the actual work.
    """

    def operation(self) -> str:
        return "Leaf"
```

```
class Composite(Component):
    """
    The Composite class represents complex components that may have children.
    It delegates the actual work to their children and then 'sum-up' the result.
    """

    def __init__(self) -> None:
        super().__init__()
        self._children: List[Component] = []

    def add(self, component: Component) -> None:
        self._children.append(component)
        component.parent = self

    def remove(self, component: Component) -> None:
        self._children.remove(component)
        component.parent = None

    def is_composite(self) -> bool:
        return True

    def operation(self) -> str:
        results = [child.operation() for child in self._children]
        return f"Branch({'+'.join(results)})"


def client_code(component: Component) -> None:
    print(f"RESULT: {component.operation()}", end="")


def client_code2(component1: Component, component2: Component) -> None:
    if component1.is_composite():
        component1.add(component2)
    print(f"RESULT: {component1.operation()}", end="")


if __name__ == "__main__":
    simple = Leaf()
    print("Client: I've got a simple component:")
    client_code(simple)
    print("\n")

    tree = Composite()

    branch1 = Composite()
    branch1.add(Leaf())
    branch1.add(Leaf())

    branch2 = Composite()
    branch2.add(Leaf())

    tree.add(branch1)
    tree.add(branch2)

    print("Client: Now I've got a composite tree:")
    client_code(tree)
    print("\n")

    print("Client: I don't need to check the components classes even when managing the tree:")
    client_code2(tree, simple)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memoize/memoize.py

```
import functools
import time

def memoize(max_size=100, timeout=None):
    def memoize_decorator(func):
        cache = {}
        timestamps = {}
        @functools.wraps(func)
        def wrapper(*args, **kwargs):
            key = (args, frozenset(kwargs.items()))

            # Check for expired cache entries
            if timeout:
                for k in list(timestamps.keys()):
                    if time.time() - timestamps[k] > timeout:
                        del cache[k]
                        del timestamps[k]

            if key in cache:
                return cache[key]

            # If cache size limit is reached, remove the oldest item
            if len(cache) >= max_size:
                oldest_key = min(timestamps, key=timestamps.get)
                del cache[oldest_key]
                del timestamps[oldest_key]

            result = func(*args, **kwargs)
            cache[key] = result
            timestamps[key] = time.time()
            return result
        return wrapper
    return memoize_decorator

# Example usage
@memoize(max_size=50, timeout=300)  # 50 items in cache and 5 minutes timeout
def some_function(arg1, arg2, **kwargs):
    # Your function implementation
    return arg1 + arg2  # Replace with actual computation

print(some_function(3, 4, option='value'))
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memoize/test_memoize.p

```
import unittest
from unittest.mock import patch
from memoize import memoize
import time

class TestMemoizeDecorator(unittest.TestCase):
    def setUp(self):
        @memoize(max_size=2, timeout=1)
        def test_func(a, b):
            return a + b
        self.test_func = test_func

    def test_basic_memoization(self):
        """Test basic memoization functionality."""
        result1 = self.test_func(1, 2)
        result2 = self.test_func(1, 2)
        self.assertEqual(result1, result2)

    def test_cache_size_limit(self):
        """Test cache size limit handling."""
        self.test_func(1, 2)
        self.test_func(3, 4)
        self.test_func(5, 6)  # This should remove the oldest cache (1, 2)
        with patch('time.time', return_value=time.time() + 2):
            result = self.test_func(1, 2)  # Recalculate as it should be removed from cache
            self.assertEqual(result, 3)

    def test_timeout_handling(self):
        """Test timeout handling in the cache."""
        self.test_func(7, 8)
        with patch('time.time', return_value=time.time() + 2):
            result = self.test_func(7, 8)  # Recalculate as it should be expired
            self.assertEqual(result, 15)
```

```
def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/memoize/__init__.py

```
from .memoize import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/iterator/__init__.py

```
from .iterator import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/iterator/iterator.py

```
from collections.abc import Iterable, Iterator
from typing import Any, List

class AlphabeticalOrderIterator(Iterator):
    """
    Concrete Iterators implement various traversal algorithms.
    """
    _position: int = None
    _reverse: bool = False

    def __init__(self, collection: List[Any], reverse: bool = False) -> None:
        self._collection = collection
        self._reverse = reverse
        self._position = -1 if reverse else 0

    def __next__(self):
        try:
            value = self._collection[self._position]
            self._position += -1 if self._reverse else 1
        except IndexError:
            raise StopIteration()
        return value

class WordsCollection(Iterable):
    """
    Concrete Collections provide methods for retrieving fresh iterator instances.
    """
    def __init__(self, collection: List[Any] = []) -> None:
        self._collection = collection

    def __iter__(self) -> AlphabeticalOrderIterator:
        return AlphabeticalOrderIterator(self._collection)

    def get_reverse_iterator(self) -> AlphabeticalOrderIterator:
        return AlphabeticalOrderIterator(self._collection, True)
    def add_item(self, item: Any):
        self._collection.append(item)

# Example usage
if __name__ == "__main__":
    collection = WordsCollection()
    collection.add_item("First")
    collection.add_item("Second")
    collection.add_item("Third")

    print("Straight traversal:")
    for item in collection:
        print(item)

    print("\nReverse traversal:")
    for item in collection.get_reverse_iterator():
        print(item)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/iterator/test_iterator.py

```
from collections.abc import Iterable, Iterator
from typing import Any, List

class AlphabeticalOrderIterator(Iterator):
    """
    Concrete Iterators implement various traversal algorithms.
    """
    _position: int = None
    _reverse: bool = False

    def __init__(self, collection: List[Any], reverse: bool = False) -> None:
        self._collection = collection
        self._reverse = reverse
        self._position = -1 if reverse else 0

    def __next__(self):
        try:
            value = self._collection[self._position]
            self._position += -1 if self._reverse else 1
        except IndexError:
            raise StopIteration()
        return value

class WordsCollection(Iterable):
    """
    Concrete Collections provide methods for retrieving fresh iterator instances.
    """
    def __init__(self, collection: List[Any] = []) -> None:
        self._collection = collection

    def __iter__(self) -> AlphabeticalOrderIterator:
        return AlphabeticalOrderIterator(self._collection)

    def get_reverse_iterator(self) -> AlphabeticalOrderIterator:
        return AlphabeticalOrderIterator(self._collection, True)

    def add_item(self, item: Any):
        self._collection.append(item)

def main():
    collection = WordsCollection()
    collection.add_item("First")
    collection.add_item("Second")
    collection.add_item("Third")

    print("Straight traversal:")
    for item in collection:
        print(item)

    print("\nReverse traversal:")
    for item in collection.get_reverse_iterator():
        print(item)

# Example usage
if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/chain_of_responsibility/__

```
from .chain_of_responsibility import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/chain_of_responsibility/tes

```
from chain_of_responsibility import AIModelHandler, DataPreprocessingHandler, VisualizationHandler, Abst

def test_individual_handler(handler: AbstractHandler, request: str):
    result = handler.handle(request)
    if result:
        print(f"  Handled by {handler.__class__.__name__}: {result}")
    else:
        print(f"  {handler.__class__.__name__} passed the request.")

def test_full_chain(chain_head: AbstractHandler, request: str):
    print(f"\nTesting full chain with request: {request}")
    result = chain_head.handle(request)
    if result:
        print(f"Handled by chain: {result}")
    else:
        print("Request was left unhandled by the full chain.")

def main():
    # Setting up individual handlers
    ai_model_handler = AIModelHandler()
    data_handler = DataPreprocessingHandler()
    visualization_handler = VisualizationHandler()

    # Building the chain
    ai_model_handler.set_next(data_handler).set_next(visualization_handler)

    # Testing individual handlers
    test_requests = ["Train", "Preprocess", "Visualize", "Deploy", "Unknown"]
    for request in test_requests:
        print(f"\nTesting AIModelHandler with request: {request}")
        test_individual_handler(ai_model_handler, request)
        print(f"\nTesting DataPreprocessingHandler with request: {request}")
        test_individual_handler(data_handler, request)
        print(f"\nTesting VisualizationHandler with request: {request}")
        test_individual_handler(visualization_handler, request)

    # Testing the full chain
    for request in test_requests:
        test_full_chain(ai_model_handler, request)

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/chain_of_responsibility/ch

```
from abc import ABC, abstractmethod
from typing import Any, Optional

class Handler(ABC):
    """
    The Handler interface declares methods for building the chain of handlers and executing requests.
    """
    @abstractmethod
    def set_next(self, handler: 'Handler') -> 'Handler':
        pass

    @abstractmethod
    def handle(self, request: Any) -> Optional[str]:
        pass

class AbstractHandler(Handler):
    """
    Default chaining behavior implementation.
    """
    _next_handler: Handler = None

    def set_next(self, handler: 'Handler') -> 'Handler':
        self._next_handler = handler
        return handler

    def handle(self, request: Any) -> Optional[str]:
        if self._next_handler:
            return self._next_handler.handle(request)
        return None

# Concrete Handlers
class AIModelHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Train":
            return f"AIModelHandler: Training model with {request}"
        else:
            return super().handle(request)

class DataPreprocessingHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Preprocess":
            return f"DataPreprocessingHandler: Preprocessing {request}"
        else:
            return super().handle(request)

class VisualizationHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Visualize":
            return f"VisualizationHandler: Visualizing data with {request}"
        else:
            return super().handle(request)

# Client code example
def client_code(handler: Handler) -> None:
    for operation in ["Train", "Preprocess", "Visualize", "Deploy"]:
        print(f"\nClient: Requesting to {operation}")
        result = handler.handle(operation)
        if result:
            print(f"  {result}", end="")
        else:
            print(f"  {operation} was left unhandled.", end="")

# Example usage
if __name__ == "__main__":
    ai_model_handler = AIModelHandler()
    data_handler = DataPreprocessingHandler()
    visualization_handler = VisualizationHandler()

    ai_model_handler.set_next(data_handler).set_next(visualization_handler)

    print("Chain: AI Model > Data Preprocessing > Visualization")
    client_code(ai_model_handler)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/visitor/__init__.py

```
from .visitor import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/visitor/visitor.py

```
from abc import ABC, abstractmethod

class Visitor(ABC):
    """
    The Visitor interface declares a set of visiting methods that correspond to
    element classes. The signature of a visiting method allows the visitor to
    identify the exact class of the element being visited.
    """
    @abstractmethod
    def visit_concrete_element_a(self, element):
        pass

    @abstractmethod
    def visit_concrete_element_b(self, element):
        pass

class ConcreteVisitor1(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"{element.operation_a()} + ConcreteVisitor1")

    def visit_concrete_element_b(self, element):
        print(f"{element.operation_b()} + ConcreteVisitor1")

class ConcreteVisitor2(Visitor):
    def visit_concrete_element_a(self, element):
        print(f"{element.operation_a()} + ConcreteVisitor2")

    def visit_concrete_element_b(self, element):
        print(f"{element.operation_b()} + ConcreteVisitor2")

class Element(ABC):
    """
    The Element interface declares an `accept` method that should take a base
    visitor interface as an argument.
    """
    @abstractmethod
    def accept(self, visitor: Visitor):
        pass

class ConcreteElementA(Element):
    def accept(self, visitor: Visitor):
        visitor.visit_concrete_element_a(self)

    def operation_a(self):
        return "ConcreteElementA"

class ConcreteElementB(Element):
    def accept(self, visitor: Visitor):
        visitor.visit_concrete_element_b(self)

    def operation_b(self):
        return "ConcreteElementB"

# Example usage
if __name__ == "__main__":
    elements = [ConcreteElementA(), ConcreteElementB()]

    visitor1 = ConcreteVisitor1()
    for element in elements:
        element.accept(visitor1)

    visitor2 = ConcreteVisitor2()
    for element in elements:
        element.accept(visitor2)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/visitor/test_visitor.py

```
import unittest
from unittest.mock import patch
from visitor import ConcreteVisitor1, ConcreteVisitor2, ConcreteElementA, ConcreteElementB

class TestConcreteVisitor1(unittest.TestCase):
    @patch('sys.stdout')
    def test_visit_concrete_element_a(self, mock_stdout):
        element_a = ConcreteElementA()
        visitor1 = ConcreteVisitor1()
        element_a.accept(visitor1)
        mock_stdout.write.assert_called_with("ConcreteElementA + ConcreteVisitor1\n")

    @patch('sys.stdout')
    def test_visit_concrete_element_b(self, mock_stdout):
        element_b = ConcreteElementB()
        visitor1 = ConcreteVisitor1()
        element_b.accept(visitor1)
        mock_stdout.write.assert_called_with("ConcreteElementB + ConcreteVisitor1\n")

class TestConcreteVisitor2(unittest.TestCase):
    @patch('sys.stdout')
    def test_visit_concrete_element_a(self, mock_stdout):
        element_a = ConcreteElementA()
        visitor2 = ConcreteVisitor2()
        element_a.accept(visitor2)
        mock_stdout.write.assert_called_with("ConcreteElementA + ConcreteVisitor2\n")

    @patch('sys.stdout')
    def test_visit_concrete_element_b(self, mock_stdout):
        element_b = ConcreteElementB()
        visitor2 = ConcreteVisitor2()
        element_b.accept(visitor2)
        mock_stdout.write.assert_called_with("ConcreteElementB + ConcreteVisitor2\n")

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/singleton/test_singleton.py

```
# test_singleton.py

from singleton import AIModelManager

def test_singleton_instance_creation():
    """
    Test that the Singleton instance is created only once.
    """
    print("Testing Singleton instance creation...")
    first_instance = AIModelManager()
    second_instance = AIModelManager()

    assert first_instance is second_instance, "Singleton instances are not the same"
    print("PASS: Singleton instance creation test")

def test_singleton_configuration_persistence():
    """
    Test that changes in configuration are reflected across all instances.
    """
    print("Testing Singleton configuration persistence...")
    manager = AIModelManager()
    initial_config = manager.get_config("response_length")
    # Change configuration
    manager.update_config("response_length", 512)

    # Create new instance and check if the configuration change is reflected
    new_manager = AIModelManager()
    new_config = new_manager.get_config("response_length")

    assert new_config == 512, "Configuration change is not reflected in the new instance"
    assert initial_config != new_config, "Initial and new configurations are the same"
    print("PASS: Singleton configuration persistence test")


def main():
    test_singleton_instance_creation()
    test_singleton_configuration_persistence()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/singleton/__init__.py

```
from .singleton import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/singleton/singleton.py

```
from threading import Lock

class SingletonMeta(type):
    """
    Thread-safe implementation of Singleton for managing AI model configurations.
    """
    _instances = {}
    _lock: Lock = Lock()

    def __call__(cls, *args, **kwargs):
        with cls._lock:
            if cls not in cls._instances:
                instance = super().__call__(*args, **kwargs)
                cls._instances[cls] = instance
        return cls._instances[cls]

class AIModelManager(metaclass=SingletonMeta):
    def __init__(self):
        # Initialize with default configuration
        self.config = {
            "language_model": "GPT-3",
            "response_length": 128,
            "custom_behavior": {}
        }
    def update_config(self, key, value):
        self.config[key] = value

    def get_config(self, key):
        return self.config.get(key, None)

    def perform_ai_logic(self):
        # Method to perform AI-related operations
        pass

# Example of direct usage:
# ai_manager = AIModelManager()
# ai_manager.update_config("response_length", 256)
# print(ai_manager.get_config("response_length"))
# ai_manager.perform_ai_logic()

# Example of indirect usage:
def main():
    # Creating the Singleton instance
    ai_manager = AIModelManager()

    # Initial configuration
    print("Initial Configuration:", ai_manager.config)

    # Updating configuration in one part of the system
    ai_manager.update_config("response_length", 256)
    print("Updated Configuration after first change:", ai_manager.config)

    # Accessing the Singleton in another part of the system
    another_manager_instance = AIModelManager()
    print("Configuration accessed from a different part:", another_manager_instance.config)

    # Demonstrating that the configuration change is reflected across all instances
    another_manager_instance.update_config("language_model", "Custom AI Model")
    print("Configuration after updating from another part:", ai_manager.config)

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/command/test_command.

```
    from command import SimpleCommand, ComplexCommand, Receiver, Invoker

    def test_simple_command():
        print("Testing SimpleCommand:")
        simple_command = SimpleCommand("Simple Operation")
        simple_command.execute()
```

```
def test_complex_command():
    print("\nTesting ComplexCommand:")
    receiver = Receiver()
    complex_command = ComplexCommand(receiver, "Data A", "Data B")
    complex_command.execute()

def test_invoker():
    print("\nTesting Invoker:")
    invoker = Invoker()
    receiver = Receiver()

    # Setting up SimpleCommand and ComplexCommand for the invoker
    invoker.set_on_start(SimpleCommand("Initialization"))
    invoker.set_on_finish(ComplexCommand(receiver, "Finalize Operation", "Clean Up"))

    invoker.do_something_important()

def main():
    test_simple_command()
    test_complex_command()
    test_invoker()

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/command/command.py

```
from abc import ABC, abstractmethod

class Command(ABC):
    """
    The Command interface declares a method for executing a command.
    """
    @abstractmethod
    def execute(self) -> None:
        pass

class SimpleCommand(Command):
    """
    Some commands can implement simple operations on their own.
    """
    def __init__(self, payload: str) -> None:
        self._payload = payload

    def execute(self) -> None:
        print(f"SimpleCommand: Doing something simple like printing ({self._payload})")
```

```
class ComplexCommand(Command):
    """
    Complex commands delegate operations to other objects, called 'receivers.'
    """
    def __init__(self, receiver: 'Receiver', a: str, b: str) -> None:
        self._receiver = receiver
        self._a = a
        self._b = b

    def execute(self) -> None:
        print("ComplexCommand: Delegating complex tasks to a receiver object")
        self._receiver.do_something(self._a)
        self._receiver.do_something_else(self._b)

class Receiver:
    """
    The Receiver class contains important business logic.
    """
    def do_something(self, a: str) -> None:
        print(f"Receiver: Working on ({a}).")

    def do_something_else(self, b: str) -> None:
        print(f"Receiver: Also working on ({b}).")

class Invoker:
    """
    The Invoker is associated with commands and sends requests to the command.
    """
    _on_start = None
    _on_finish = None

    def set_on_start(self, command: Command):
        self._on_start = command

    def set_on_finish(self, command: Command):
        self._on_finish = command

    def do_something_important(self) -> None:
        print("Invoker: Does anybody want something done before I begin?")
        if isinstance(self._on_start, Command):
            self._on_start.execute()

        print("\nInvoker: ...doing something really important...")

        print("\nInvoker: Does anybody want something done after I finish?")
        if isinstance(self._on_finish, Command):
            self._on_finish.execute()
```

```
# Example usage
if __name__ == "__main__":
    invoker = Invoker()
    invoker.set_on_start(SimpleCommand("Start operation"))
    receiver = Receiver()
    invoker.set_on_finish(ComplexCommand(receiver, "Send email", "Save report"))

    invoker.do_something_important()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/command/__init__.py

```
from .command import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/flyweight/flyweight.py

```
import json
from typing import Dict, List

class Flyweight:
    def __init__(self, shared_state: List[str]) -> None:
        self._shared_state = shared_state

    def operation(self, unique_state: List[str]) -> None:
        shared = json.dumps(self._shared_state)
        unique = json.dumps(unique_state)
        print(f"Flyweight: Shared ({shared}) and unique ({unique}) state.")

class FlyweightFactory:
    _flyweights: Dict[str, Flyweight] = {}

    def __init__(self, initial_flyweights: List[List[str]]) -> None:
        for state in initial_flyweights:
            self._flyweights[self.get_key(state)] = Flyweight(state)

    def get_key(self, state: List[str]) -> str:
        return "_".join(sorted(state))

    def get_flyweight(self, shared_state: List[str]) -> Flyweight:
        key = self.get_key(shared_state)
        if not self._flyweights.get(key):
            print("FlyweightFactory: Creating new flyweight.")
            self._flyweights[key] = Flyweight(shared_state)
        else:
            print("FlyweightFactory: Reusing existing flyweight.")
        return self._flyweights[key]

    def list_flyweights(self) -> None:
        print(f"FlyweightFactory: I have {len(self._flyweights)} flyweights:")
        for key in self._flyweights:
            print(key)

# Client code example
def add_ai_component_to_system(factory: FlyweightFactory, data: List[str]) -> None:
    flyweight = factory.get_flyweight(data[:-1])
    flyweight.operation(data)

# Example usage
if __name__ == "__main__":
    factory = FlyweightFactory([
        ["NeuralNet", "Classifier", "Image"],
        ["NeuralNet", "Regressor", "TimeSeries"]
    ])

    factory.list_flyweights()

    add_ai_component_to_system(factory, ["NeuralNet", "Classifier", "Image", "ImageSetA"])
    add_ai_component_to_system(factory, ["NeuralNet", "Classifier", "Audio", "AudioSetB"])

    factory.list_flyweights()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/flyweight/__init__.py

```
    from .flyweight import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/flyweight/test_flyweight.py

```
    from flyweight import FlyweightFactory, add_ai_component_to_system

    def test_flyweight_pattern():
        # Creating a Flyweight Factory with some initial shared states
        factory = FlyweightFactory([
            ["NeuralNet", "Classifier", "Image"],
            ["NeuralNet", "Regressor", "TimeSeries"]
        ])

        # Listing initial flyweights
        print("Initial flyweights in the factory:")
        factory.list_flyweights()

        # Adding components and testing if flyweights are reused or newly created
        print("\nAdding a new AI component to the system:")
        add_ai_component_to_system(factory, ["NeuralNet", "Classifier", "Image", "DatasetX"])

        print("\nAdding another AI component to the system (should reuse flyweight):")
        add_ai_component_to_system(factory, ["NeuralNet", "Classifier", "Image", "DatasetY"])

        print("\nAdding a different AI component (should create new flyweight):")
        add_ai_component_to_system(factory, ["NeuralNet", "Classifier", "Audio", "DatasetZ"])
        # Listing final flyweights to verify the correct creation and reuse
        print("\nFinal flyweights in the factory:")
        factory.list_flyweights()

    def main():
        test_flyweight_pattern()

    if __name__ == "__main__":
        main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/builder/__init__.py

```
from .builder import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/builder/builder.py

```
from abc import ABC, abstractmethod

# Builder Interface
class AIComponentBuilder(ABC):

    @property
    @abstractmethod
    def product(self):
        """Property to get the product."""
        pass

    @abstractmethod
    def add_ai_module(self):
        """Method to add an AI module to the product."""
        pass

    @abstractmethod
    def add_learning_capability(self):
        """Method to add learning capability to the product."""
        pass

    @abstractmethod
    def add_interaction_interface(self):
        """Method to add an interaction interface to the product."""
        pass

# Concrete Builder
class ConcreteAIComponentBuilder(AIComponentBuilder):
    def __init__(self):
        self.reset()

    def reset(self):
        """Reset the builder to start with a fresh product."""
        self._product = AIComponent()
    @property
    def product(self):
        """Retrieve the built product and reset the builder."""
        product = self._product
        self.reset()
        return product

    def add_ai_module(self):
        """Add an AI module to the product."""
        self._product.add("AI Module")

    def add_learning_capability(self):
        """Add learning capability to the product."""
        self._product.add("Learning Capability")

    def add_interaction_interface(self):
        """Add an interaction interface to the product."""
        self._product.add("Interaction Interface")

# Product Class
class AIComponent:
    def __init__(self):
        self.parts = []

    def add(self, part):
        """Add a part to the component."""
        self.parts.append(part)

    def list_parts(self):
        """List all parts of the component."""
        print(f"AI Component Parts: {', '.join(self.parts)}", end="")

# Director Class
class Director:
    def __init__(self):
        self._builder = None

    @property
    def builder(self):
        """Property to get and set the builder."""
        return self._builder

    @builder.setter
    def builder(self, builder):
        self._builder = builder
    def build_minimal_ai_component(self):
        """Build a minimal AI component."""
        self.builder.add_ai_module()

    def build_full_featured_ai_component(self):
        """Build a full-featured AI component."""
        self.builder.add_ai_module()
        self.builder.add_learning_capability()
        self.builder.add_interaction_interface()

# Client Code (optional here, might be in a separate test file)
if __name__ == "__main__":
    director = Director()
    builder = ConcreteAIComponentBuilder()
    director.builder = builder

    print("Building a minimal AI component:")
    director.build_minimal_ai_component()
    builder.product.list_parts()

    print("\n\nBuilding a full-featured AI component:")
    director.build_full_featured_ai_component()
    builder.product.list_parts()

    print("\n\nBuilding a custom AI component:")
    builder.add_ai_module()
    builder.add_interaction_interface()
    builder.product.list_parts()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/builder/test_builder.py

```
from builder import Director, ConcreteAIComponentBuilder

def test_minimal_ai_component(director, builder):
    print("Testing minimal AI component construction:")
    director.builder = builder
    director.build_minimal_ai_component()
    builder.product.list_parts()

def test_full_featured_ai_component(director, builder):
    print("\nTesting full-featured AI component construction:")
    director.builder = builder
    director.build_full_featured_ai_component()
    builder.product.list_parts()

def test_custom_ai_component(builder):
    print("\nTesting custom AI component construction:")
    builder.add_ai_module()
    builder.add_learning_capability()  # Adding only specific parts
    builder.product.list_parts()

def main():
    director = Director()
    builder = ConcreteAIComponentBuilder()

    test_minimal_ai_component(director, builder)
    test_full_featured_ai_component(director, builder)
    test_custom_ai_component(builder)

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/mediator/test_mediator.py

```
import unittest
from unittest.mock import patch
from mediator import Mediator, ConcreteMediator, BaseComponent, Component1, Component2

class TestMediatorPattern(unittest.TestCase):
    def setUp(self):
        self.component1 = Component1()
        self.component2 = Component2()
        self.mediator = ConcreteMediator(self.component1, self.component2)

    def test_mediator_initialization(self):
        """Test if the mediator is correctly set in the components."""
        self.assertEqual(self.component1.mediator, self.mediator)
        self.assertEqual(self.component2.mediator, self.mediator)

    def test_component_communication(self):
        """Test the communication between components via the mediator."""
        with patch('sys.stdout') as mock_stdout:
            self.component1.do_a()
            self.assertIn("Component 1 does A.", mock_stdout.getvalue())
            self.assertIn("Mediator reacts on A and triggers:", mock_stdout.getvalue())
            self.assertIn("Component 2 does C.", mock_stdout.getvalue())

            mock_stdout.reset()
            self.component2.do_d()
            self.assertIn("Component 2 does D.", mock_stdout.getvalue())
            self.assertIn("Mediator reacts on D and triggers:", mock_stdout.getvalue())
            self.assertIn("Component 1 does B.", mock_stdout.getvalue())
            self.assertIn("Component 2 does C.", mock_stdout.getvalue())

    def test_mediator_reactions(self):
        """Test mediator's reactions to different events."""
        with patch('sys.stdout') as mock_stdout:
            self.mediator.notify(self.component1, "A")
            self.assertIn("Mediator reacts on A and triggers:", mock_stdout.getvalue())
            self.assertIn("Component 2 does C.", mock_stdout.getvalue())

            mock_stdout.reset()
            self.mediator.notify(self.component2, "D")
            self.assertIn("Mediator reacts on D and triggers:", mock_stdout.getvalue())
            self.assertIn("Component 1 does B.", mock_stdout.getvalue())
            self.assertIn("Component 2 does C.", mock_stdout.getvalue())

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/mediator/mediator.py

```
from abc import ABC, abstractmethod

class Mediator(ABC):
    """
    The Mediator interface declares a method for components to notify the mediator about events.
    """
    @abstractmethod
    def notify(self, sender: object, event: str) -> None:
        pass

class ConcreteMediator(Mediator):
    def __init__(self, component1: 'Component1', component2: 'Component2') -> None:
        self._component1 = component1
        self._component1.mediator = self
        self._component2 = component2
        self._component2.mediator = self

    def notify(self, sender: object, event: str) -> None:
        if event == "A":
            print("Mediator reacts on A and triggers:")
            self._component2.do_c()
        elif event == "D":
            print("Mediator reacts on D and triggers:")
            self._component1.do_b()
            self._component2.do_c()

class BaseComponent:
    """
    Base Component class with a mediator.
    """
    def __init__(self, mediator: Mediator = None) -> None:
        self._mediator = mediator

    @property
    def mediator(self) -> Mediator:
        return self._mediator

    @mediator.setter
    def mediator(self, mediator: Mediator) -> None:
        self._mediator = mediator

class Component1(BaseComponent):
    def do_a(self) -> None:
        print("Component 1 does A.")
        self.mediator.notify(self, "A")

    def do_b(self) -> None:
        print("Component 1 does B.")
        self.mediator.notify(self, "B")

class Component2(BaseComponent):
    def do_c(self) -> None:
        print("Component 2 does C.")
        self.mediator.notify(self, "C")

    def do_d(self) -> None:
        print("Component 2 does D.")
        self.mediator.notify(self, "D")

# Example usage
if __name__ == "__main__":
    c1 = Component1()
    c2 = Component2()
    mediator = ConcreteMediator(c1, c2)

    print("Client triggers operation A.")
    c1.do_a()

    print("\nClient triggers operation D.")
    c2.do_d()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/mediator/__init__.py

```
from .mediator import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/factory/test_factory.py

```
from factory import BotFactory, ChatBot, DataAnalysisBot, ConcurrentBot
```

```
def test_bot_creation(factory):
    # Testing the creation of ChatBot
    print("Testing ChatBot Creation:")
    chat_bot = factory.create_bot("chat")
    print(chat_bot.perform_task())

    # Testing the creation of DataAnalysisBot
    print("\nTesting DataAnalysisBot Creation:")
    data_bot = factory.create_bot("data")
    print(data_bot.perform_task())

def test_dynamic_bot_registration(factory):
    # Dynamically registering and testing a new bot type
    class ResearchBot:
        def perform_task(self):
            return "ResearchBot performing research."

    factory.register_new_bot_type("research", ResearchBot)
    research_bot = factory.create_bot("research")
    print("\nTesting Dynamically Registered ResearchBot:")
    print(research_bot.perform_task())

def main():
    bot_factory = BotFactory()
    bot_factory.register_new_bot_type("chat", ChatBot)
    bot_factory.register_new_bot_type("data", DataAnalysisBot)
    bot_factory.register_new_bot_type("concurrent", lambda: ConcurrentBot(ChatBot()))

    test_bot_creation(bot_factory)
    test_dynamic_bot_registration(bot_factory)

if __name__ == "__main__":
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/factory/__init__.py

```
from .factory import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/factory/factory.py

```
from abc import ABC, abstractmethod
import threading

# Step 1: Dynamic Bot Factory Interface
class Factory(ABC):
    @abstractmethod
    def create_bot(self, bot_type):
        pass

    @abstractmethod
    def register_new_bot_type(self, bot_type, bot_creator):
        pass

# Step 2: Polymorphic Concrete Bot Factories
class BotFactory(Factory):
    def __init__(self):
        self.bot_creators = {}

    def create_bot(self, bot_type):
        return self.bot_creators[bot_type]()

    def register_new_bot_type(self, bot_type, bot_creator):
        self.bot_creators[bot_type] = bot_creator

# Step 3: Abstract Bot Interface
class AbstractBot(ABC):
    @abstractmethod
    def perform_task(self):
        pass

    @abstractmethod
    def learn_new_skill(self, skill):
        pass

# Step 4: Various AI Bots
class ChatBot(AbstractBot):
    def perform_task(self):
        return "ChatBot engaging in conversation."

    def learn_new_skill(self, skill):
        return f"ChatBot learning {skill}."

class DataAnalysisBot(AbstractBot):
    def perform_task(self):
        return "DataAnalysisBot analyzing data."

    def learn_new_skill(self, skill):
        return f"DataAnalysisBot learning {skill}."

# Step 5: Concurrency in Bots
class ConcurrentBot(AbstractBot):
    def __init__(self, bot):
        self.bot = bot
        self.lock = threading.Lock()

    def perform_task(self):
        with self.lock:
            return self.bot.perform_task()

    def learn_new_skill(self, skill):
        with self.lock:
            return self.bot.learn_new_skill(skill)

# Step 6: Client Code Demonstration
def client_code(factory: BotFactory):
    factory.register_new_bot_type("chat", ChatBot)
    factory.register_new_bot_type("data", DataAnalysisBot)

    chat_bot = factory.create_bot("chat")
    print(chat_bot.perform_task())

    # Dynamically registering a new bot type
    factory.register_new_bot_type("concurrent", lambda: ConcurrentBot(ChatBot()))
    concurrent_bot = factory.create_bot("concurrent")
    print(concurrent_bot.perform_task())

# Demonstration
if __name__ == "__main__":
    bot_factory = BotFactory()
    client_code(bot_factory)
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/strategy/test_strategy.py

```
import unittest
from unittest.mock import patch
from strategy import Context, ConcreteStrategyA, ConcreteStrategyB, Strategy, reverse_alphabetical

class TestStrategyPattern(unittest.TestCase):
    def test_concrete_strategy_a(self):
        """Test ConcreteStrategyA."""
        strategy = ConcreteStrategyA()
        data = ["e", "b", "d", "a", "c"]
        result = strategy.do_algorithm(data)
        self.assertEqual(result, ["a", "b", "c", "d", "e"])

    def test_concrete_strategy_b(self):
        """Test ConcreteStrategyB."""
        strategy = ConcreteStrategyB()
        data = ["e", "b", "d", "a", "c"]
        result = list(strategy.do_algorithm(data))
        self.assertEqual(result, ["e", "d", "c", "b", "a"])

    def test_context_with_different_strategies(self):
        """Test the context with different strategies."""
        context = Context(ConcreteStrategyA())
        data = ["e", "b", "d", "a", "c"]

        with patch('sys.stdout') as mock_stdout:
            context.do_some_business_logic()
            self.assertIn(','.join(sorted(data)), mock_stdout.getvalue())

        context.strategy = ConcreteStrategyB()
        with patch('sys.stdout') as mock_stdout:
            context.do_some_business_logic()
            self.assertIn(','.join(reversed(sorted(data))), mock_stdout.getvalue())

        context.strategy = Strategy(lambda data: reverse_alphabetical(data))
        with patch('sys.stdout') as mock_stdout:
            context.do_some_business_logic()
            self.assertIn(','.join(reversed(sorted(data))), mock_stdout.getvalue())

def main():
    unittest.main()

if __name__ == '__main__':
    main()
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/strategy/__init__.py

```
from .strategy import *
```

/Users/ctavolazzi/Code/WinfoNova/Nova_System_Git/NovaSystem/DesignPatterns/strategy/strategy.py

```
from abc import ABC, abstractmethod
from typing import List, Callable, Any
from dataclasses import dataclass

class Strategy(ABC):
    @abstractmethod
    def do_algorithm(self, data: List[str]) -> List[str]:
        pass

class ConcreteStrategyA(Strategy):
    def do_algorithm(self, data: List[str]) -> List[str]:
        return sorted(data)

class ConcreteStrategyB(Strategy):
    def do_algorithm(self, data: List[str]) -> List[str]:
        return reversed(sorted(data))

# Example of a functional strategy using a lambda function
reverse_alphabetical = lambda data: reversed(sorted(data))

@dataclass
class Context:
    strategy: Strategy

    def do_some_business_logic(self) -> None:
        result = self.strategy.do_algorithm(["a", "b", "c", "d", "e"])
        print(",".join(result))

# Example usage
if __name__ == "__main__":
    context = Context(ConcreteStrategyA())
    print("Client: Strategy is set to normal sorting.")
    context.do_some_business_logic()

    print("\nClient: Strategy is set to reverse sorting.")
    context.strategy = ConcreteStrategyB()
    context.do_some_business_logic()

    # Using a functional strategy
    print("\nClient: Strategy is set to functional reverse sorting.")
    context.strategy = Strategy(lambda data: reverse_alphabetical(data))
    context.do_some_business_logic()
```