UNIVERSITY OF MICHIGAN - DEARBORN
CIS 476 - SOFTWARE ARCHITECTURE & DESIGN PATTERNS

TERM PROJECT (MY PASS)

TEAM MEMBERS:

AHMAD HASHEM
JAD JONAIDI
MOHAMAD SYAJ

AGENDA

1. Program Demonstration
2. Architecture Overview
3. Design Patterns Used
4. Individual Code Explanations
5. Task Summary
6. GitHub & Video Submission

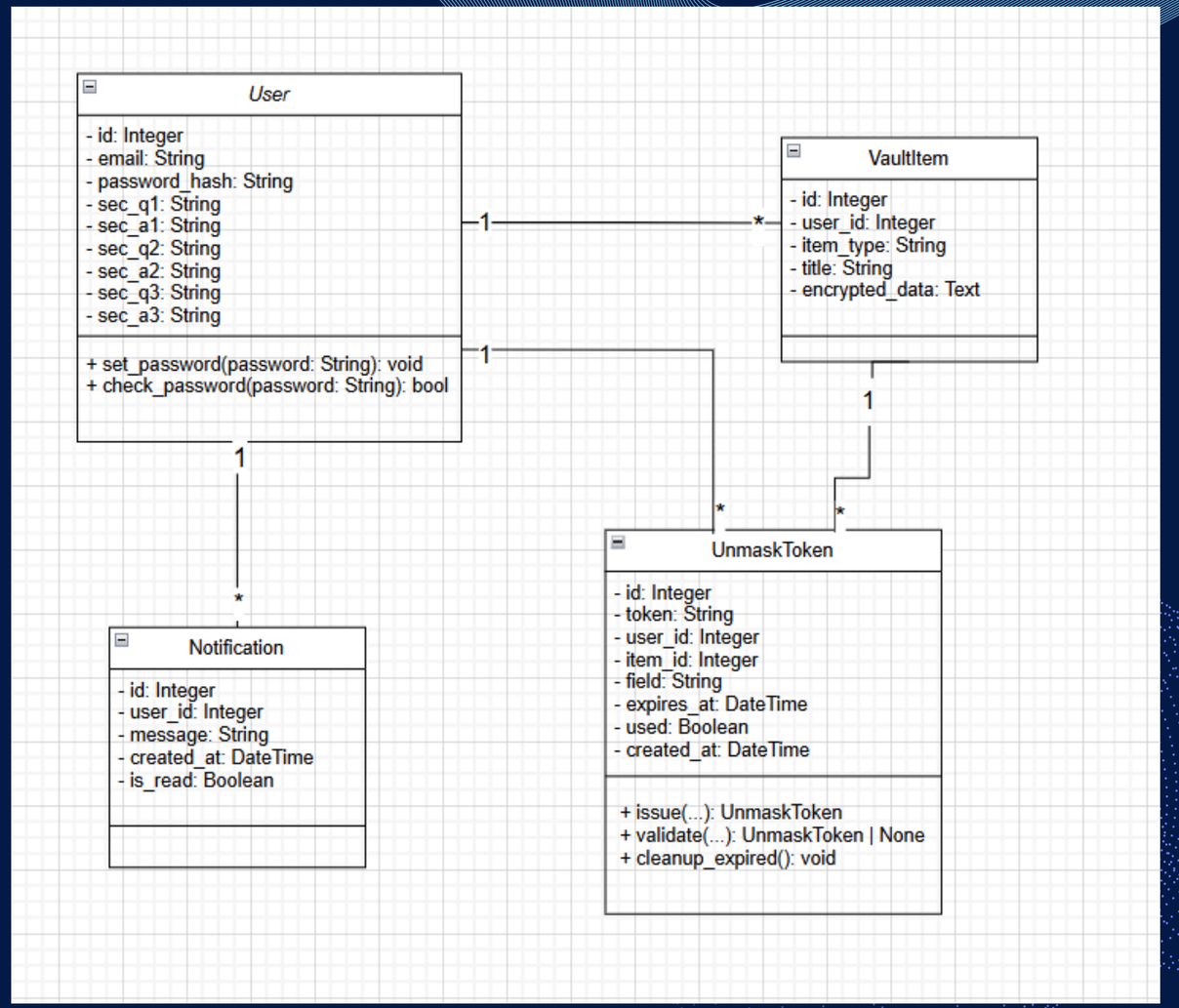
PROGRAM DEMONSTRATION

“Welcome to our project demonstration. Our application is a secure password manager built in Flask with SQLAlchemy, implementing multiple design patterns.
We’ll begin by running the app and walking through the main features.”

Presentation

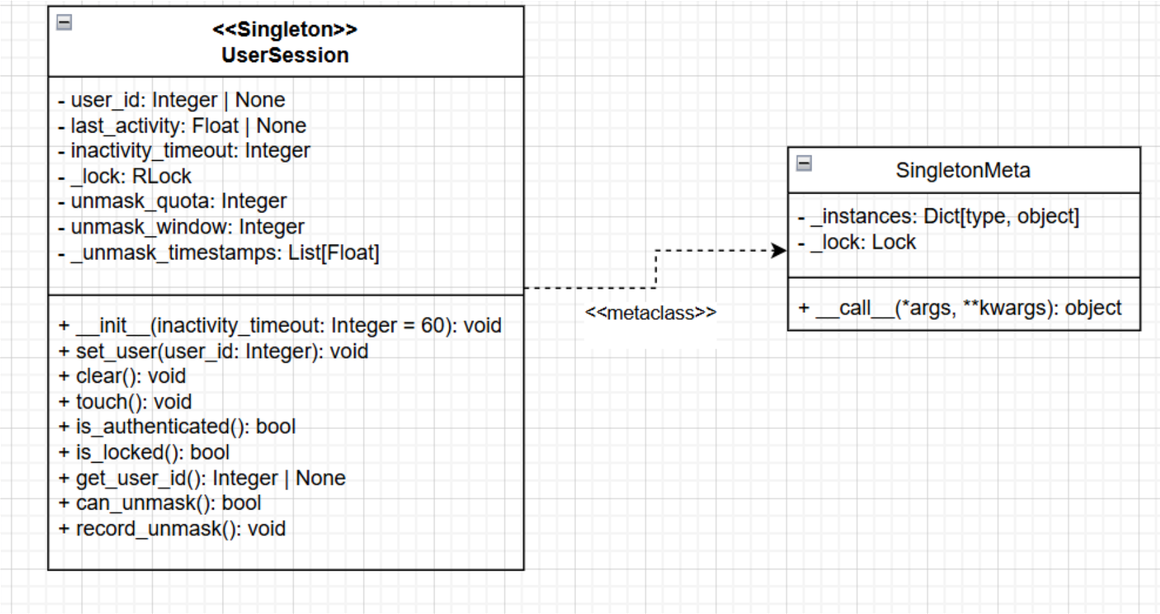
SYSTEM ARCHITECTURE OVERVIEW

- Modular Flask app using SQLAlchemy
- Secure token-based unmasking
- Centralized session manager
- Notification system
- UI-mediator future-ready architecture

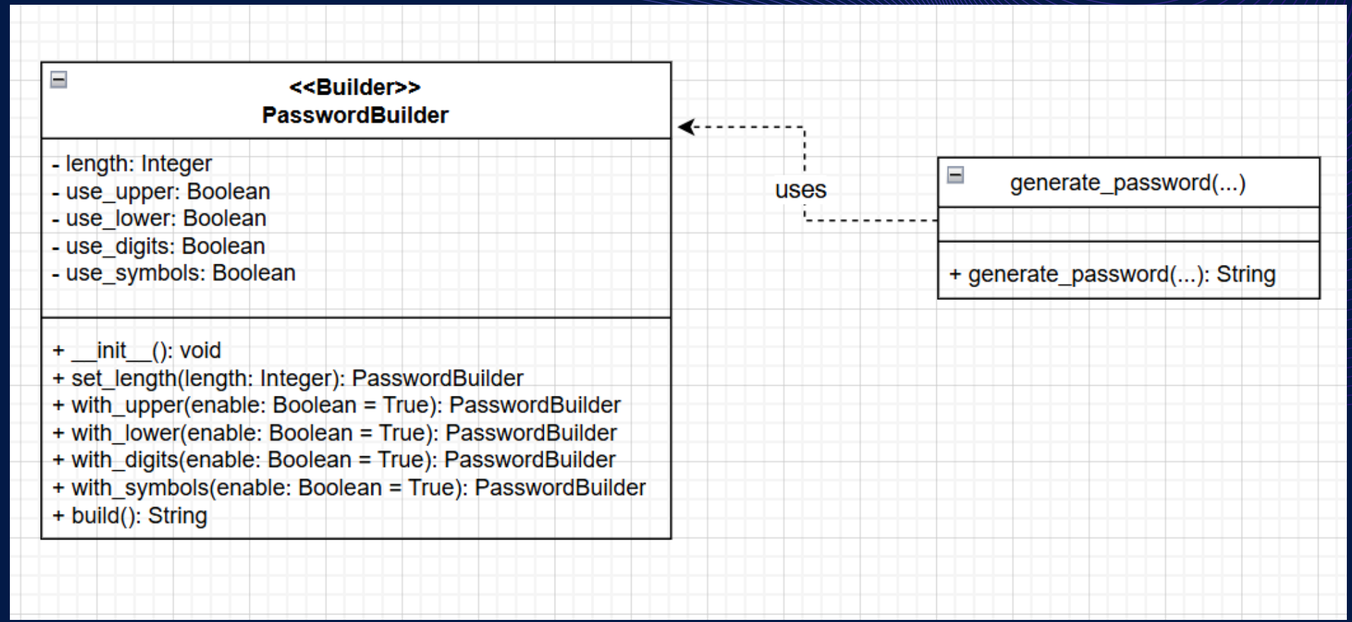


DESIGN PATTERNS USED

SINGLETON - USERSESSION

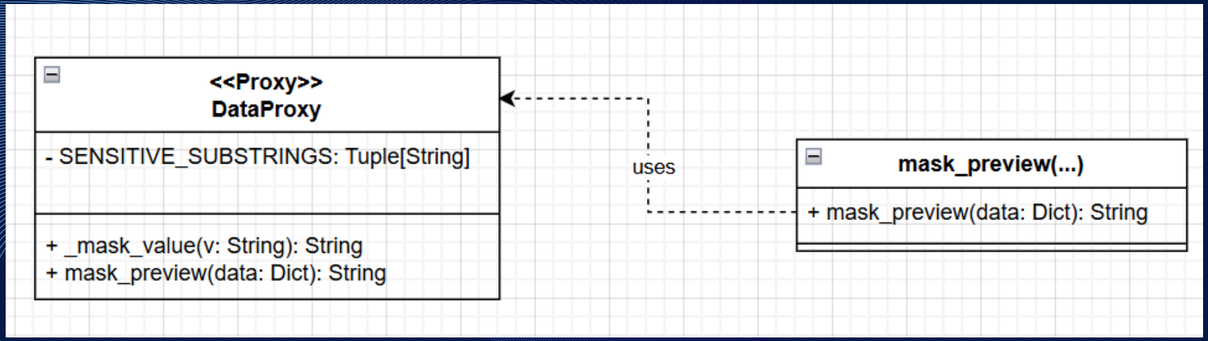


BUILDER - PASSWORDBUILDER

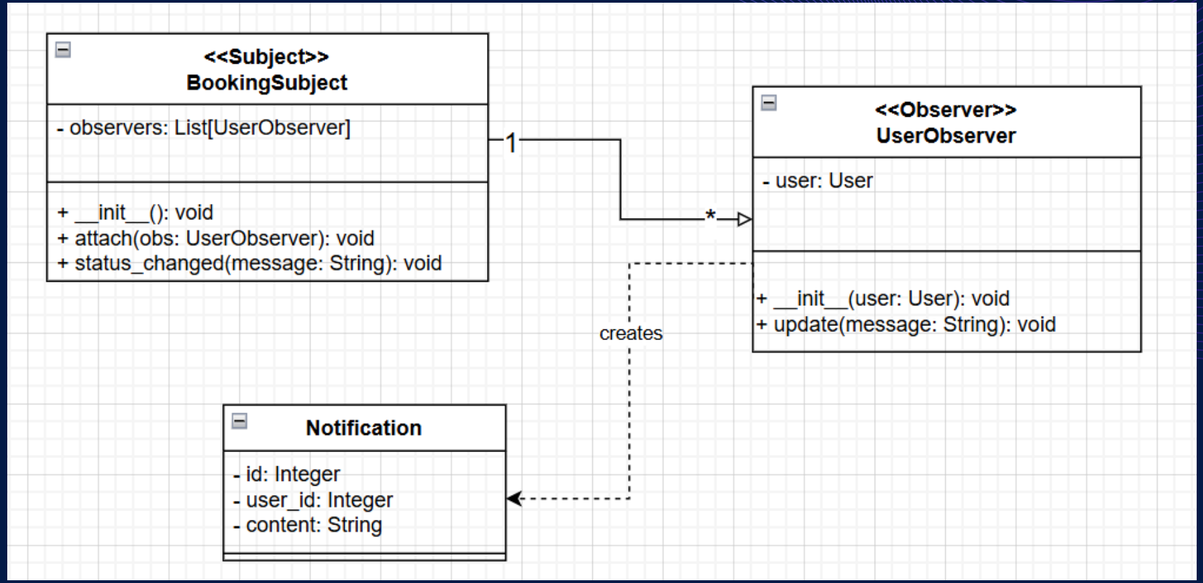


DESIGN PATTERNS USED

PROXY - DATAPROXY (MASKED PREVIEWS)

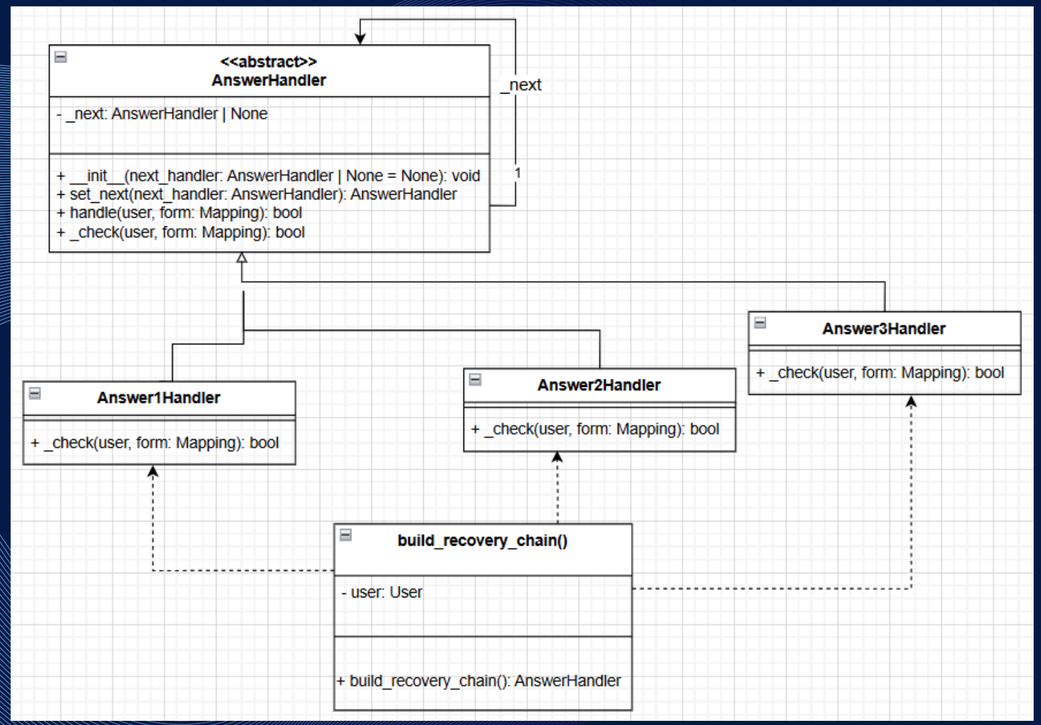


- ## OBSERVER - BOOKINGSUBJECT + USEROBSERVER (NOTIFICATIONS)

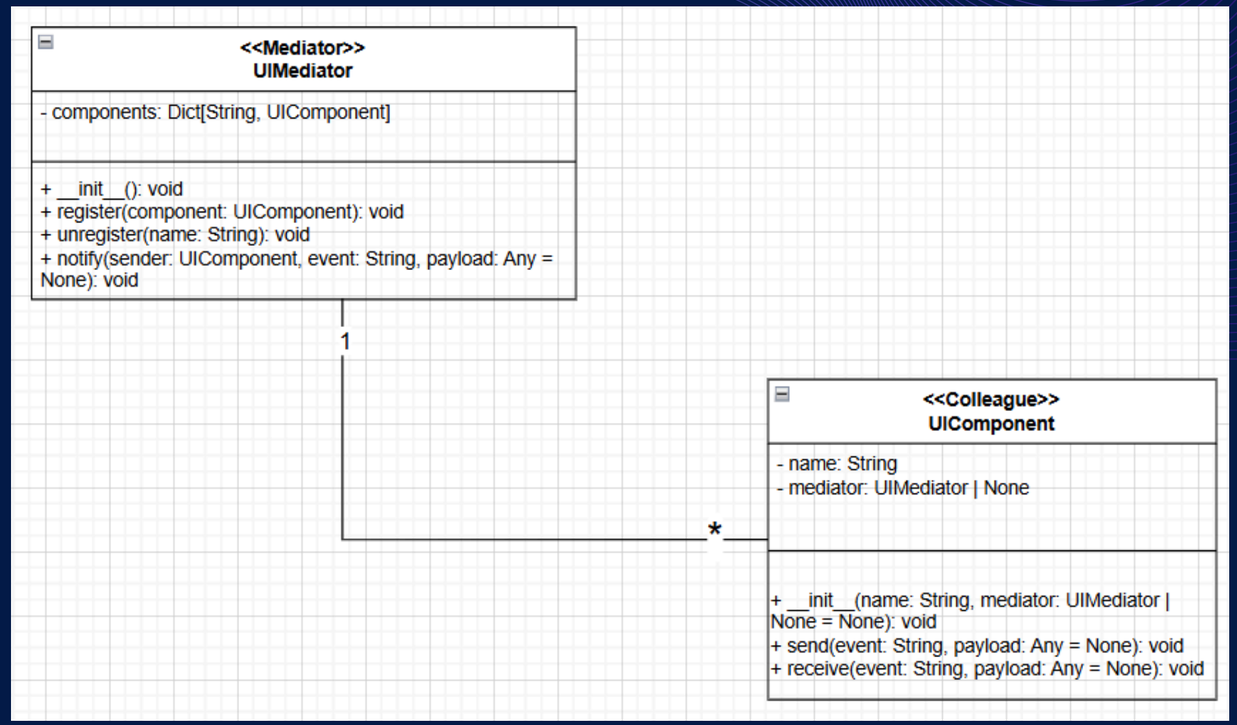


DESIGN PATTERNS USED

CHAIN OF RESPONSIBILITY - ANSWERHANDLERS FOR SECURITY QUESTIONS



MEDIATOR - UIMEDIATOR COORDINATING COMPONENTS



SINGLETON PATTERN (USER SESSION MANAGER)

Location: patterns/singleton.py

Purpose:
Ensures that the application uses only one session manager instance throughout the entire runtime.

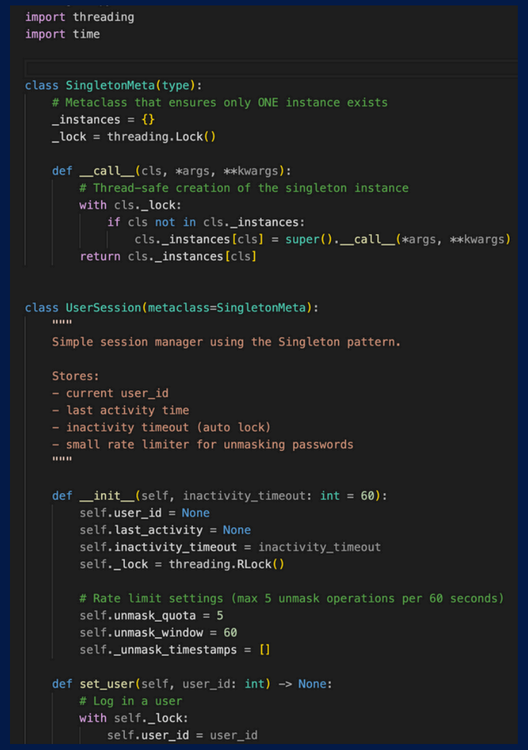
Used For:

- Tracking the currently logged-in user
- Managing inactivity timeouts
- Rate-limiting unmask/copy operations
- Preventing duplicate session objects

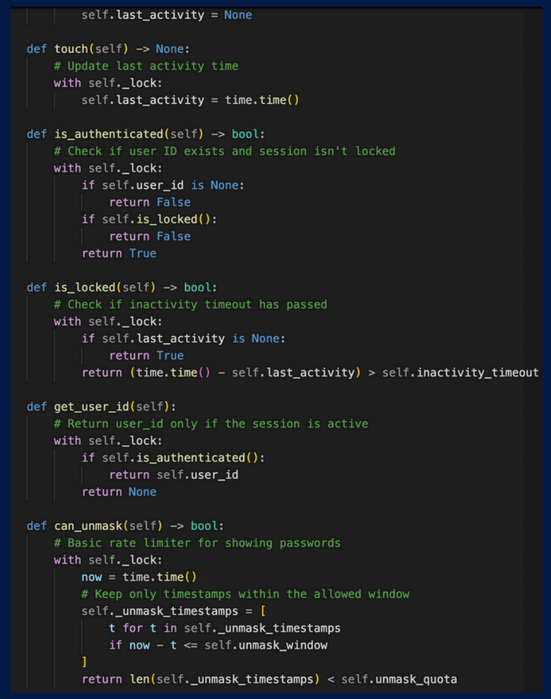
Key Components:

- SingletonMeta metaclass
- UserSession (the Singleton)

Why we used it:
Centralizing user state avoids conflicting session instances and improves security.



```python
import threading
import time


class SingletonMeta(type):
    # Metaclass that ensures only ONE instance exists
    _instances = {}
    _lock = threading.Lock()

    def __call__(cls, *args, **kwargs):
        # Thread-safe creation of the singleton instance
        with cls._lock:
            if cls not in cls._instances:
                cls._instances[cls] = super().__call__(*args, **kwargs)
        return cls._instances[cls]


class UserSession(metaclass=SingletonMeta):
    """
    Simple session manager using the Singleton pattern.

    Stores:
    - current user_id
    - last activity time
    - inactivity timeout (auto lock)
    - small rate limiter for unmasking passwords
    """

    def __init__(self, inactivity_timeout: int = 60):
        self.user_id = None
        self.last_activity = None
        self.inactivity_timeout = inactivity_timeout
        self._lock = threading.RLock()

        # Rate limit settings (max 5 unmask operations per 60 seconds)
        self.unmask_quota = 5
        self.unmask_window = 60
        self._unmask_timestamps = []

    def set_user(self, user_id: int) -> None:
        # Log in a user
        with self._lock:
            self.user_id = user_id
```



```python
        self.last_activity = None

    def touch(self) -> None:
        # Update last activity time
        with self._lock:
            self.last_activity = time.time()

    def is_authenticated(self) -> bool:
        # Check if user ID exists and session isn't locked
        with self._lock:
            if self.user_id is None:
                return False
            if self.is_locked():
                return False
            return True

    def is_locked(self) -> bool:
        # Check if inactivity timeout has passed
        with self._lock:
            if self.last_activity is None:
                return True
            return (time.time() - self.last_activity) > self.inactivity_timeout

    def get_user_id(self):
        # Return user_id only if the session is active
        with self._lock:
            if self.is_authenticated():
                return self.user_id
            return None

    def can_unmask(self) -> bool:
        # Basic rate limiter for showing passwords
        with self._lock:
            now = time.time()
            # Keep only timestamps within the allowed window
            self._unmask_timestamps = [
                t for t in self._unmask_timestamps
                if now - t <= self.unmask_window
            ]
            return len(self._unmask_timestamps) < self.unmask_quota
```

BUILDER PATTERN (PASSWORD GENERATION)

Location: patterns/password_builder.py

Purpose:
Builds complex password strings by selecting character sets and length through chainable methods.

Used For:

- Generating strong, customizable passwords

Key Components:

- PasswordBuilder
- build() method
- Helper function generate_password(...)

Why we used it:
Builder makes password generation flexible, readable, and easy to extend.

```python
import secrets
import string
from typing import Optional


class PasswordBuilder:
    def __init__(self):
        # Default password settings
        self.length = 16
        self.use_upper = True
        self.use_lower = True
        self.use_digits = True
        self.use_symbols = False

    def set_length(self, length: int) -> 'PasswordBuilder':
        # Make sure length isn't too small
        self.length = max(4, int(length))
        return self

    def with_upper(self, enable: bool = True) -> 'PasswordBuilder':
        # Turn uppercase letters on/off
        self.use_upper = bool(enable)
        return self

    def with_lower(self, enable: bool = True) -> 'PasswordBuilder':
        # Turn lowercase letters on/off
        self.use_lower = bool(enable)
        return self

    def with_digits(self, enable: bool = True) -> 'PasswordBuilder':
        # Turn digits on/off
        self.use_digits = bool(enable)
        return self

    def with_symbols(self, enable: bool = True) -> 'PasswordBuilder':
        # Turn special characters on/off
        self.use_symbols = bool(enable)
        return self
```

```python
    def with_symbols(self, enable: bool = True) -> 'PasswordBuilder':
        # Turn special characters on/off
        self.use_symbols = bool(enable)
        return self

    def build(self) -> str:
        # Build the character pool based on chosen options
        pool = ''
        if self.use_lower:
            pool += string.ascii_lowercase
        if self.use_upper:
            pool += string.ascii_uppercase
        if self.use_digits:
            pool += string.digits
        if self.use_symbols:
            pool += '!@#$%^&*()-_=+[]{};:,.<>?'

        # If everything is disabled, fall back to letters + digits
        if not pool:
            pool = string.ascii_letters + string.digits

        # Make sure we include at least one character from each enabled type
        password_chars = []
        classes = []
        if self.use_lower:
            classes.append(string.ascii_lowercase)
        if self.use_upper:
            classes.append(string.ascii_uppercase)
        if self.use_digits:
            classes.append(string.digits)
        if self.use_symbols:
            classes.append('!@#$%^&*()-_=+[]{};:,.<>?')

        # Add one guaranteed character from each class
        for cls in classes:
            password_chars.append(secrets.choice(cls))

        # Fill the rest randomly until we reach the desired length
        while len(password_chars) < self.length:
            password_chars.append(secrets.choice(pool))
```

PROXY PATTERN (MASKED DATA PREVIEW)

Location: patterns/data_proxy.py

Purpose:
Controls access to sensitive vault fields and provides safe, masked previews.

Used For:

- Showing items in the dashboard without exposing full passwords or card numbers

Key Components:

- DataProxy class
- _mask_value()
- mask_preview()
- mask_preview_dict()

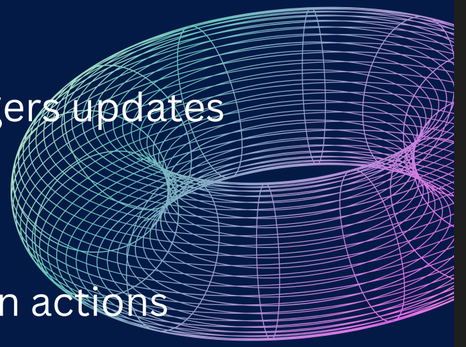
Why we used it:
A proxy prevents accidental leaks of sensitive information and ensures safe data handling.

```python
class DataProxy:

    # Words that indicate the field is sensitive and should be masked
    SENSITIVE_SUBSTRINGS = ('password', 'card', 'cvv', 'ssn', 'social', 'passport', 'license')

    def _mask_value(self, v: str) -> str:
        # Convert value to string safely
        s = '' if v is None else str(v)

        # If value is short, mask the whole thing
        if len(s) <= 4:
            return '*' * len(s)

        # Otherwise keep first and last character visible
        return s[0] + '*' * (len(s) - 2) + s[-1]

    def mask_preview(self, data: dict) -> str:
        # Return empty string for empty input
        if not data:
            return ''

        parts = []
        for k, v in data.items():
            key = (k or '').lower()

            # If the key looks sensitive, mask the value
            if any(sub in key for sub in self.SENSITIVE_SUBSTRINGS):
                parts.append(f"{k}: {self._mask_value(v)}")
            else:
                parts.append(f"{k}: {v}")

        # Join everything into a single readable string
        return '; '.join(parts)

def mask_preview(data: dict) -> str:
    # Helper wrapper for string preview
    return DataProxy().mask_preview(data)

def mask_preview_dict(data: dict) -> dict:
    # Return empty dict if nothing is provided
    if not data:
        return {}
```

SLIDE 9 - OBSERVER PATTERN (NOTIFICATIONS)

Location: patterns/observer.py

Purpose:
A notification system that automatically triggers updates when specific events occur.



Used For:

- Creating Notification records when certain actions happen (password recovery events)

Key Components:

- BookingSubject (subject)
- UserObserver (observer)
- Notification model (generated output)

Why we used it:
Observer decouples event broadcasting from notification creation, making the system easier to extend.

```python
from models.notification import Notification
from models import db


class BookingSubject:
    def __init__(self):
        # List of all observers (users who want updates)
        self.observers = []

    def attach(self, obs):
        # Add a new observer
        self.observers.append(obs)

    def status_changed(self, message):
        # Notify every observer about the update
        for o in self.observers:
            o.update(message)


class UserObserver:
    def __init__(self, user):
        # Store the user this observer represents
        self.user = user

    def update(self, message):
        # Create a notification record for this user
        try:
            note = Notification(user_id=self.user.id, content=message)
            db.session.add(note)
            db.session.commit()
        except Exception:
            # If something fails, undo the database changes
            db.session.rollback()
```

CHAIN OF RESPONSIBILITY (SECURITY QUESTION VALIDATION)

Location: patterns/chain_of_responsibility.py

Purpose:
Validates security-question answers in sequential steps, passing the request along the chain.

Used For:

- Password recovery flow
- Checking Question #1, then #2, then #3
- Stopping on first failure

Key Components:

- AnswerHandler (abstract)
- Answer1Handler
- Answer2Handler
- Answer3Handler
- build_recovery_chain()

Why we used it:
CoR removes nested if-statements and makes each answer-check independent and composable.

```python
from __future__ import annotations

from abc import ABC, abstractmethod
import hmac
from typing import Optional, Mapping, Any


def _normalize(value: Optional[str]) -> str:
    # Clean the string: make lowercase and remove spaces
    return (value or "").strip().lower()


class AnswerHandler(ABC):
    # Base class for checking one security answer

    def __init__(self, next_handler: Optional["AnswerHandler"] = None) -> None:
        # Points to the next checker in the chain
        self._next = next_handler

    def set_next(self, next_handler: "AnswerHandler") -> "AnswerHandler":
        # Set who comes next in the chain
        self._next = next_handler
        return next_handler

    def handle(self, user, form: Mapping[str, Any]) -> bool:
        # First check this answer
        if not self._check(user, form):
            return False

        # Then move to the next handler if it exists
        if self._next:
            return self._next.handle(user, form)

        return True

    @abstractmethod
    def _check(self, user, form: Mapping[str, Any]) -> bool:
        # Each child class must implement its own check
        ...
```

```python
class Answer1Handler(AnswerHandler):
    def _check(self, user, form: Mapping[str, Any]) -> bool:
        # Compare user's answer 1 with the stored correct answer
        a1 = _normalize(form.get("a1"))
        correct1 = _normalize(getattr(user, "sec_a1", None))
        return hmac.compare_digest(a1, correct1)


class Answer2Handler(AnswerHandler):
    def _check(self, user, form: Mapping[str, Any]) -> bool:
        a2 = _normalize(form.get("a2"))
        correct2 = _normalize(getattr(user, "sec_a2", None))
        return hmac.compare_digest(a2, correct2)


class Answer3Handler(AnswerHandler):
    def _check(self, user, form: Mapping[str, Any]) -> bool:
        a3 = _normalize(form.get("a3"))
        correct3 = _normalize(getattr(user, "sec_a3", None))
        return hmac.compare_digest(a3, correct3)


def build_recovery_chain() -> AnswerHandler:
    # Build the chain: answer1 → answer2 → answer3
    first = Answer1Handler()
    second = first.set_next(Answer2Handler())
    second.set_next(Answer3Handler())
    return first


def verify_security_answers(user, form: Mapping[str, Any]) -> bool:
    # Run the full chain of answer checks
    chain = build_recovery_chain()
    return chain.handle(user, form)
```

MEDIATOR PATTERN (UI COMPONENT COORDINATION)

Location: patterns/mediator.py

Purpose:
Decouples UI components so they communicate through a mediator instead of directly.

Used For:

Broadcasting events between components

Setting groundwork for interactive UI updates

Key Components:

UIMediator

UIComponent (colleague)

send() and receive()

```python
from typing import Dict, Any, Optional

class UIComponent:
    def __init__(self, name: str, mediator: Optional['UIMediator'] = None):
        self.name = name
        self.mediator = mediator  # The mediator this component talks through

    def send(self, event: str, payload: Any = None):
        # Tell the mediator that something happened
        if self.mediator:
            self.mediator.notify(self, event, payload)

    def receive(self, event: str, payload: Any = None):
        # Other components will override this
        pass

class UIMediator:
    def __init__(self):
        # Holds all registered UI components
        self._components: Dict[str, UIComponent] = {}

    def register(self, component: UIComponent) -> None:
        # Add component and connect it to this mediator
        self._components[component.name] = component
        component.mediator = self

    def unregister(self, name: str) -> None:
        # Remove component and disconnect it
        comp = self._components.pop(name, None)
        if comp:
            comp.mediator = None

    def notify(self, sender: UIComponent, event: str, payload: Any = None) -> None:
        # Send the event to every component except the sender
        for name, comp in self._components.items():
            if comp is sender:
                continue
            try:
                comp.receive(event, payload)
            except Exception:
                # Ignore errors from components
                pass
```

INDIVIDUAL CONTRIBUTIONS

Ahmad Hashem:

- Frontend templates (HTML/UI)
- Auth flow UI + routing
- Singleton Pattern (UserSession)
- Mediator Pattern (UI event coordination)
- Unmask feature UI integration

Jad Jonaidi:

- Chain of Responsibility (security questions)
- Builder Pattern (password generator)
- Database models & schema
- Recovery flow logic
- System design diagrams

Mohamad Syaj:

- Proxy Pattern (masked previews)
- Observer Pattern (notifications)
- Vault item & unmask backend logic
- GitHub management / repo maintenance
- Preview masking UI behavior

Task Summary Table

Feature / Task	Ahmad Hashem	Jad Jonaidi	Mohamad Syaj
Frontend Templates	✓	✓	✓
Auth / Session System	✓		
Singleton Pattern	✓		
Chain of Responsibility		✓	
Proxy Pattern			✓
Builder Pattern		✓	
Observer Pattern			✓
Mediator Pattern	✓		
Models / DB		✓	
GitHub + Docs			✓
Slides / Video Editing	✓	✓	✓

GitHub & Video Submission

GitHub Repository:
https://github.com/mohamadsyaj/CIS476-final-Project.git

Video Presentation (Pre-recorded):
https://umich-my.sharepoint.com/:v:/g/personal/mbsyaj_umich_edu/ESLXI-V7U6RBtwqsIKnJJt0BEBuq2XelV46nzPGYsmwdqw?nav=eyJyZWZlcnJhbEluZm8iOnsicmVmZXJyYWxBcHAiOiJPbmVEcml2ZUZvckJ1c2luZXNzIiwicmVmZXJyYWxBcHBQbGF0Zm9ybSI6IldlYiIsInJlZmVycmFsTW9kZSI6InZpZXciLCJyZWZlcnJhbFZpZXciOiJNeUZpbGVzTGlua0NvcHkifX0&e=YffLmu

Team:
Ahmad Hashem
Jad Jonaidi
Mohamad Syaj

THANK YOU