Criteria C

Product Development

Programming Techniques	Evidence of Use (page)	Purpose
OOP	MainWindow Class, SettingsPage class, SplashScreen Class, ModelTrainer class Pages:	OOP allows me to modularize my code, keeping functionalities separate, improving maintainability. Each window is encapsulated in its own class, making my application more scalable. Aggregation and inheritance are also used to reduce redundancy in code.
Aggregation	MainWindow Class: ● Aggregates multiple components like timers, ML models, and UI elements. ● Manages a collection of settings and categories. This enables the class to manage everything related to the main functionality of the app.	Aggregation helps me with modular design and allows my components to be reused without great redundancy. **Success Criteria 1**: Helps manage the logic needed for classifying user activity. **Success Criteria 4**: Aggregates different components required to monitor user activity.
Encapsulation	My MainWindow class encapsulates settings, user-defined categories, and machine learning models. ● Attributes: settings, vectorizer, model, categories, etc. ● Methods: load_settings(), save_settings(), detect_unproductive_activity() My SettingsPage class encapsulates the UI and logic for modifying user settings. ● Attributes: sessions, OverrideDelay, WarningMessage ● Methods: add_session(), save_settings()	Encapsulation makes the code more modular and easier to manage or extend. **Success Criteria 9**: Encapsulation of settings ensures user changes are saved and managed effectively.
Inheritance	MainWindow(QMainWindow, Ui_MainWindow) and SettingsPage(QMainWindow, Ui_Settings) both inherit from QMainWindow, gaining all the functionalities of a GUI window in PyQt.	Inheritance allows me to focus on implementing specific features in classes and windows without rewriting common window functionality. **Success Criteria 2**: Helps in creating customizable UI elements for user-defined categories.
Iteration	In check_sessions(), self.sessions is iterated over to determine if the current time falls within any user-defined session. This ensures that the program only runs when it should. In populate_unproductive_categories(), self.category_sites.items() is iterated over to	Iteration is useful when needing to program a repetitive task. Oftentimes, iteration means this repetitive task is written in a shorter code. **Success Criteria 2**: Populating UI

	populate UI components dynamically with categories and examples.	components with user-selected unproductive categories. **Success Criteria 3**: Iteration over user-defined session times ensures the program runs only when it should.
Dynamic Behaviour	The functions load_settings() and save_settings() are used to dynamically read and write user preferences. This allows the program to adapt to user changes without hardcoding values. The method start_training() dynamically retrains the machine learning model based on user preferences.	Allows the app to adapt to user changes without requiring code modifications. **Success Criteria 5, 7, 9**: Allows for real-time adaptation to user preferences, such as delay times and intervention messages.
Lists and Dictionaries	Lists like self.whitelisted_sites and self.blacklisted_sites are used to store and manage URLs. Sessions is a list of dictionaries where each dictionary represents a session in which the program can run, with fields for start_time, end_time, and days. Category_sites is a dictionary where keys are category names and the values are lists of example websites for each category. A dictionary is ideal for key-value pairs, as it is efficient for updates or lookups.	Lists and dictionaries are flexible ways to store items, allowing for iteration over elements. Their dynamic nature is helpful for manipulating data. **Success Criteria 2, 3, 4, 9**: Efficient management of categories and sessions enables smooth operation of the application.
Error Handling	On page 4, I have evidence of error handling. In the examples, I used try-except blocks to handle errors. One case was if the settings file is missing upon load or read, after catching the exception the program would use settings on a pre-set template. I also found that HTTP requests and parsing website content were particularly susceptible to errors, whether due to network errors or invalid URLs. I catch these exceptions to ensure the program continues to run even if a website cannot be fetched.	Error Handling is crucial in ensuring that the program can handle unexpected situations or failures without breaking. It helps me provide a user-friendly experience by preventing crashes to the program.
Logging	On page 5, I have evidence of logging. I've shown my use of print messages for logging, showing the process of detecting unproductive activity and the status of the machine learning model. By doing so, I can know whether the program is running correctly or not and receive feedback on what process is occurring, assisting me with debugging as well.	Logging is essential to help me debug and understand how my program is behaving.

Configuration File	On page 6, I show the code of my configuration file. User preferences like session times, categories selected as unproductive, and delay times are saved in the settings.json file. At the start of the program, I load the settings from the file, and when changes are made during runtime, settings are updated.	I created a settings.json Configuration File to help meet my success criteria of saving user changes and settings. It allows user preferences to persist across sessions of using the program without having to code them into the application or unnecessarily setting up a database, which takes up more storage and space compared to the lightweight JSON format. **Success Criteria 9:** Saves user preferences
Meaningful Variable Names	I gave variables names such as wait_time, sessions, unproductive_flag, warning_displayed, that clearly indicate their purpose: wait_time stores how long the program should wait before displaying a warning, unproductive_flag tracks whether unproductive activity has been detected, and warning_displayed stores whether a warning message has just been or is currently being displayed.	Meaningful Variable names make the code easier to read, understand, and maintain/debug. By using meaningful variable names, the purpose of the variables is clear, preventing confusion and helping me with the maintenance and debugging of the code.

Dependencies	**Purpose**
sys	To control the exit of the program (sys.exit()), as the sys module provides access to functions and objects to interact with the Python runtime environment.
json	To read and write user settings to a file (settings.json), allowing user preference to persist.
PyQt5	To create GUI for the program: used classes from PyQt5 for functions such as tracking time, managing multiple threads for different tasks, e.t.c
TfidVectorizer	Part of sckit-learn's feature extraction module, used to convert text to numerical features that will be used by the machine learning model to classify content in productive and unproductive.
LogisticRegression	A machine learning algorithm from sckit-learn, used to predict whether extracted text from the screen is productive or unproductive.
ImageGrab	A module from Python Imaging Library, used to take screenshots of the user's display.
pytesseract	Python function of Google's Tesseract Optical Character Recognition engine, used to extract text from screenshots captured by ImageGrab.
requests	To fetch content of websites.
BeautifulSoup	To extract text from the content of websites fetched using requests, which is then used to train the machine learning model to make predictions.
re	To extract only words from the website content, helps generate a clean input for the machine learning model.
os	To set environment variables related to PyQt's display to allow for better scaling and display on screens.

SplashScreen Class

[Success criteria 1: Displays a loading screen and starts training the ML model to allow for detection of productive/unproductive activity]

```
# Class for the SplashScreen which is displayed while the program is loading
class SplashScreen(QMainWindow):
    def __init__(self):
        QMainWindow.__init__(self)
        self.ui = Ui_SplashScreen()  # Initialize the splash screen UI
        self.ui.setupUi(self)
        self.center()  # Center the splash screen on the screen

        # Make the window frameless and translucent for a modern look
        self.setWindowFlag(Qt.FramelessWindowHint)
        self.setAttribute(Qt.WA_TranslucentBackground)

        self.counter = 0  # Counter for the progress bar

        # Timer to update the progress bar every 50 milliseconds
        self.timer = QtCore.QTimer()
        self.timer.timeout.connect(self.progress)
        self.timer.start(50)  # Start the timer

        self.show()  # Show the splash screen

        # Prepare the main window but do not show it yet
        self.main_window = MainWindow()

        # Start training the machine learning model in a separate thread
        self.model_trainer = ModelTrainer(self.main_window)
        self.model_trainer.training_finished.connect(self.on_training_finished)
        self.model_trainer.start()  # Start the model training
```

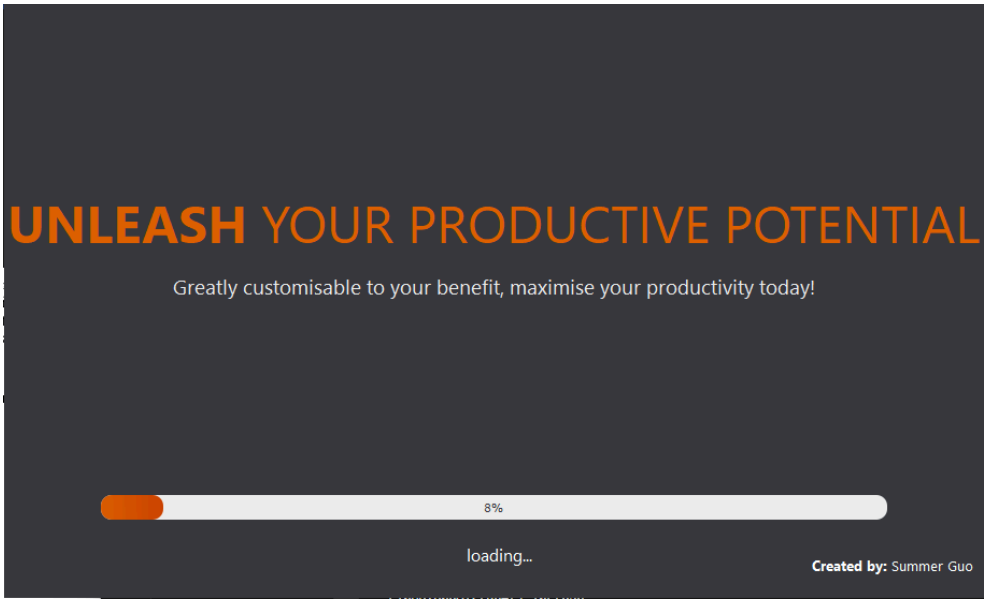
Default constructors __init__(self), center, progress, on_training_finished. Constructors improve the readability of my code by making it clear what values are being initialised and how. They are also used to enforce encapsulation by ensuring that the object's attributes are initialized correctly and in a controlled manner. ("Constructors in Python," 2024) By encapsulating attributes such as self.counter and self.model_trainer, the internal state and data of this class is only accessible via defined methods, improving safety and abstraction.

Inheritance of QMainWindow allows SplashScreen to gain all the functionalities of a GUI window in PyQt. For example, window behaviours such as minimise and maximise are automatically inherited. Aggregation of ModelTrainer and QtCore.QTimer allows me to reduce redundancy in writing the same code multiple times. Also, by aggregating model training to another class, training the model can occur on another thread and not block UI.

Defining private attributes such as self.counter in the __init__ method helps abstract data, making the code more secure and easier to manage.

```
# Function to center the splash screen on the display
def center(self):
    qr = self.frameGeometry()  # Get the geometry of the window
    cp = QDesktopWidget().availableGeometry().center()  # Get the center of the screen
    qr.moveCenter(cp)  # Move the window to the center
    self.move(qr.topLeft())  # Move the window to the top-left corner of the geometry

# Update the progress bar and eventually close the splash screen
def progress(self):
    self.ui.progressBar.setValue(self.counter)  # Set progress bar value

    if self.counter > 100:  # If the counter exceeds 100, close the splash screen
        self.timer.stop()  # Stop the timer
        self.main_window.showMaximized()  # Show the main window maximized
        self.close()  # Close the splash screen

    self.counter += 1  # Increment the counter

# Called when the model training is finished
def on_training_finished(self):
    print("Model training completed.")
    self.main_window.on_training_finished()  # Notify the main window that training is done
```

SplashScreen GUI



ModelTrainer Class

[Success criteria 1]

```
# ModelTrainer class runs model training in the background on a separate thread
class ModelTrainer(QThread):
    training_finished = pyqtSignal()  # Signal emitted when training is complete

    def __init__(self, main_window):
        super().__init__()
        self.main_window = main_window  # Reference to the main window that holds the model

    # This method is run when the thread starts
    def run(self):
        self.main_window.train_model()  # Call the main window's training method
        print("start training")
        self.training_finished.emit()  # Emit the signal when training is finished
```

I used Multithreading to handle concurrency, allowing computationally extensive tasks to run without freezing the user interface. I train the Machine Learning model on a separate thread using the ModelTrainer class, as training can take time and freeze the UI if run on the main thread. Through Multithreading, the UI remains responsive to the user while model training runs in the background. ModelTrainer inherits from QThread, and aggregates the train_model() method from the MainWindow class.

MainWindow class

[Success Criteria 1, 2, 4: Selectable unproductive category choices that inform productive/unproductive activity detection]

Machine Learning (Logistic Regression)

```
# Initialize the machine learning model for detecting unproductive activity
self.vectorizer = TfidfVectorizer()  # Convert text to numeric features using TF-IDF
self.model = LogisticRegression(max_iter=1000)  # Use logistic regression with a high iteration limit
```

```
# Predict whether the given text is unproductive using the trained model
def predict_unproductive(self, text):
    if not self.currently_in_session or not self.model_trained:
        return False  # If not in session or model isn't trained, return False

    content_features = self.vectorizer.transform([text])  # Transform the text into numeric features
    prediction = self.model.predict(content_features)  # Use the model to make a prediction
    return prediction[0] == "unproductive"  # Return True if the prediction is "unproductive"
```

Logistic Regression is an algorithm specifically designed for binary classifications (i.e. productive or unproductive), and helps me meet my success criteria of classification of user activity into productive or unproductive. Compared to other Machine Learning models like Support Vector Machines or Neural Networks, Logistic Regression has lower computational cost, meaning it's faster to train and predict.

The constructor predict_unproductive abstracts attributes such as content_features and prediction.

```python
# Train the machine learning model based on user preferences and example sites
def train_model(self):
    data = []
    labels = []

    # Use example sites from the selected unproductive categories to train the model
    for category, examples in self.category_sites.items():
        if self.categories.get(category):
            for site in examples:
                content = self.fetch_website_content("http://" + site)
                if content:
                    data.append(content)
                    labels.append("unproductive")
        else:
            for site in examples:
                content = self.fetch_website_content("http://" + site)
                if content:
                    data.append(content)
                    labels.append("productive")

    # Add whitelisted and blacklisted sites to the training data
    for site in self.whitelisted_sites:
        content = self.fetch_website_content("http://" + site)
        if content:
            data.append(content)
            labels.append("productive")

    for site in self.blacklisted_sites:
        content = self.fetch_website_content("http://" + site)
        if content:
            data.append(content)
```

```python
# Train the model using the collected data
features = self.vectorizer.fit_transform(data)  # Transform the data into features
self.model.fit(features, labels)  # Train the logistic regression model
```

In the train_model constructor, data and labels are used to train the Logistic Regression model. The text content of websites used to train the model is first converted into numerical features using TfidfVectorizer. I then used conditional statements with iteration to input all the websites and content as data to train the model.

Optical Character Recognition: Tesseract OCR

```python
# Extract text from an image using Tesseract OCR
def extract_text_from_image(self, image):
    try:
        text = pytesseract.image_to_string(image)
        return text
    except pytesseract.TesseractNotFoundError:
        print("Tesseract is not found in the path used")
        return ""
    except Exception as e:
        print(f"{e}")
        return ""
```

OCR functionality is encapsulated in the class, isolating the logic for text extraction, and this modularity helps keep my code clean and easier to maintain.
Tesseract Optical Character Recognition is a tool used to extract text from images. I use it to extract text from the screenshot of the user's screen, captured using ImageGrab. This content will then be used to either train the logistic regression machine learning algorithm or will be fed to the algorithm for classification. This helps me meet my success criteria of monitoring user activity.

Error Handling

```python
# Function to load settings from the settings file when the program starts
def load_settings():
    try:
        # Open the settings file and load the content as a dictionary
        with open(SETTINGS_FILE, 'r') as f:
            settings = json.load(f)
            # Convert session times from strings to QTime objects for easier manipulation
            for session in settings.get('sessions', []):
                if isinstance(session["start_time"], str):
                    session["start_time"] = QTime.fromString(session["start_time"], "HH:mm")
                if isinstance(session["end_time"], str):
                    session["end_time"] = QTime.fromString(session["end_time"], "HH:mm")
            return settings  # Return the loaded settings
    except FileNotFoundError:
        # If the settings file doesn't exist, return default settings
        return {
            "wait_time": "5",
            "sessions": [],
            "categories": {},
            "whitelisted_sites": [],
            "blacklisted_sites": [],
            "productive_sites": [],  # List of productive sites
            "override_delay": False,
            "warning_message": "Unproductive activity detected! For your own good, please return to being productive!"
        }
```

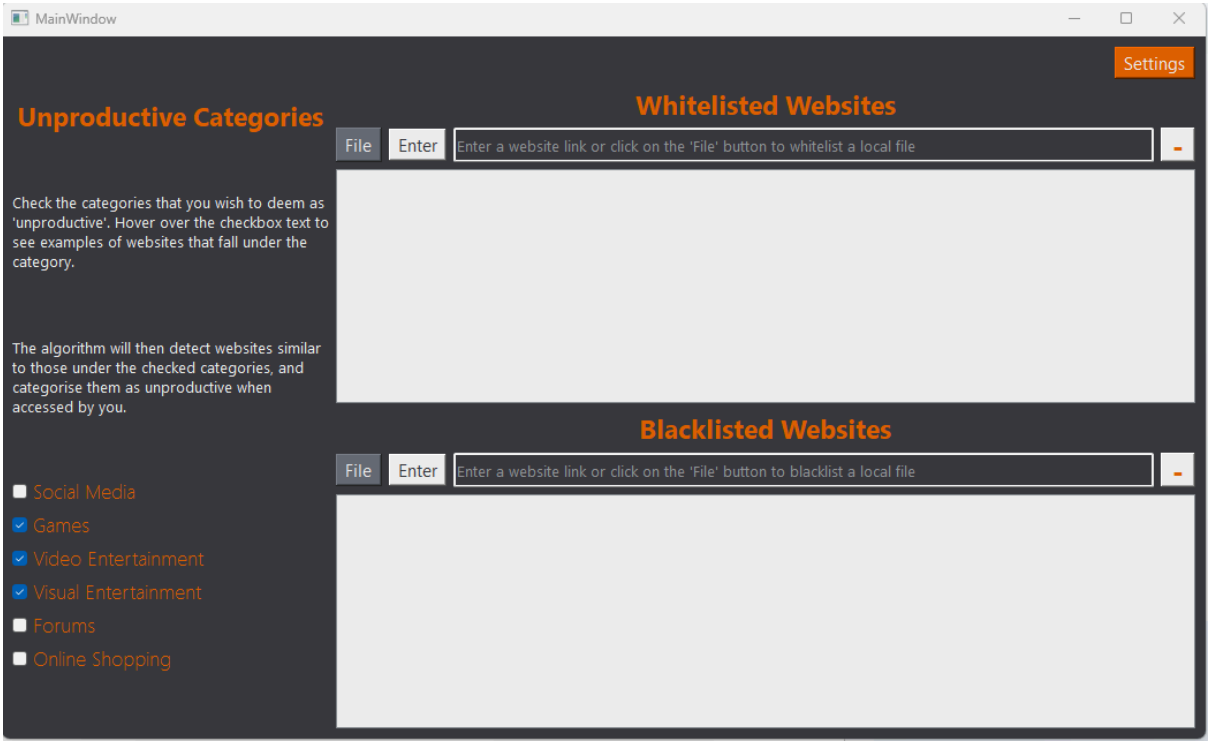
```python
# Fetch the content of a website for training or prediction
def fetch_website_content(self, url):
    if not url.startswith("http://") and not url.startswith("https://"):
        url = "http://" + url  # Ensure the URL starts with "http://"
    try:
        response = requests.get(url)  # Send a request to the website
        soup = BeautifulSoup(response.text, 'html.parser')  # Parse the HTML content
        return ' '.join(re.findall(r'\w+', soup.get_text().lower()))  # Return the cleaned text
    except Exception as e:
        print(f"Error fetching website content: {e}")
        return None  # Return None if there's an error
```

Logging

```python
# Detect unproductive activity by analyzing the text on the screen
def detect_unproductive_activity(self):
    # Only proceed if the user is in an active session and the model is trained
    if not self.currently_in_session:
        print("Current time is not within any session. Will not proceed with activity detection.")
        return

    if not self.model_trained:
        print("Model is not yet trained. Will not proceed with activity detection.")
        return

    print("Detecting unproductive activity...")
    screen_image = self.capture_screen()  # Capture the screen
    screen_text = self.extract_text_from_image(screen_image)  # Extract text from the captured image

    if screen_text.strip():
        print("Extracted text from screen:", screen_text)
        self.unproductive_flag = self.predict_unproductive(screen_text)  # Predict if the text is unproductive
        print(f"Screen text is {'unproductive' if self.unproductive_flag else 'productive'}.")

        # Handle detected unproductive activity
        if self.unproductive_flag:
            if self.override_delay:
                print("Override delay is enabled, displaying warning message immediately.")
                self.display_warning_message()  # Show warning immediately if override delay is enabled
            else:
                if not self.unproductive_timer.isValid():
                    print("Starting unproductive timer.")
                    self.unproductive_timer.start()  # Start the timer for unproductive activity
                elif self.unproductive_timer.elapsed() >= self.wait_time * 60000:
                    print(f"Unproductive timer elapsed {self.wait_time} minutes, displaying warning message.")
                    self.display_warning_message()  # Show warning if the wait time has passed
                    self.unproductive_timer.invalidate()  # Reset the timer
                else:
                    remaining_time = self.wait_time * 60000 - self.unproductive_timer.elapsed()
                    print(f"Unproductive timer running. Time remaining: {remaining_time // 60000}
                          minutes and {remaining_time % 60000 // 1000} seconds.")
```

```python
# Called when the model training is finished
def on_training_finished(self):
    print("Model training completed.")
    self.main_window.on_training_finished()  # Notify the main window that training is done
```

MainWindow GUI



SettingsPage class

[Success Criteria 3, 5, 6, 7, 9]

Conditional Statements

```
# Check if the current time falls within any of the user's work sessions
def check_sessions(self):
    current_time = QTime.currentTime()  # Get the current time
    current_day = QtCore.QDate.currentDate().dayOfWeek() - 1  # Get the current day (0=Monday, 6=Sunday)

    in_session = False  # Flag to track if the current time is within a session
    for session in self.sessions:
        start_time = session["start_time"]
        end_time = session["end_time"]

        # Check if the current day is part of the session's active days
        if session["days"][current_day]:
            if end_time < start_time:  # If the session spans midnight
                if current_time >= start_time or current_time <= end_time:
                    in_session = True
                    print(f"Current time {current_time.toString()} is
                          within session: {start_time.toString()} - {end_time.toString()} (spans midnight)")
                    break
            else:
                if start_time <= current_time <= end_time:  # If the session is during the same day
                    in_session = True
                    print(f"Current time {current_time.toString()} is
                          within session: {start_time.toString()} - {end_time.toString()}")
                    break
```

To help me control the flow of my program based on specific conditions, I used if-else conditional statements. For instance, in the above code, I check whether the current time is within a session. Since sessions can span midnight, I need to consider the different cases, where either the start time value is less than the end time value (such as start time 08:00, end time 13:00), or if the start time value is more than the end time value (such as start time 13:00, end time 01:00).

```
# Handle detected unproductive activity
if self.unproductive_flag:
    if self.override_delay:
        print("Override delay is enabled, displaying warning message immediately.")
        self.display_warning_message()  # Show warning immediately if override delay is enabled
    else:
        if not self.unproductive_timer.isValid():
            print("Starting unproductive timer.")
            self.unproductive_timer.start()  # Start the timer for unproductive activity
        elif self.unproductive_timer.elapsed() >= self.wait_time * 60000:
            print(f"Unproductive timer elapsed {self.wait_time} minutes,
                  displaying warning message.")
            self.display_warning_message()  # Show warning if the wait time has passed
            self.unproductive_timer.invalidate()  # Reset the timer
        else:
            remaining_time = self.wait_time * 60000 - self.unproductive_timer.elapsed()
            print(f"Unproductive timer running. Time remaining: {remaining_time // 60000}
                  minutes and {remaining_time % 60000 // 1000} seconds.")
else:
    print("Resetting unproductive flag and timer.")
    self.unproductive_flag = False
    self.unproductive_timer.invalidate()  # Reset the timer if the activity is productive
```

In the above, I have a nested if-else condition to check if unproductive activity has been detected (using the unproductive flag) and whether the user has enabled the ‘override delay’ option. If enabled, the warning message is shown immediately, otherwise, the program starts a timer to check whether

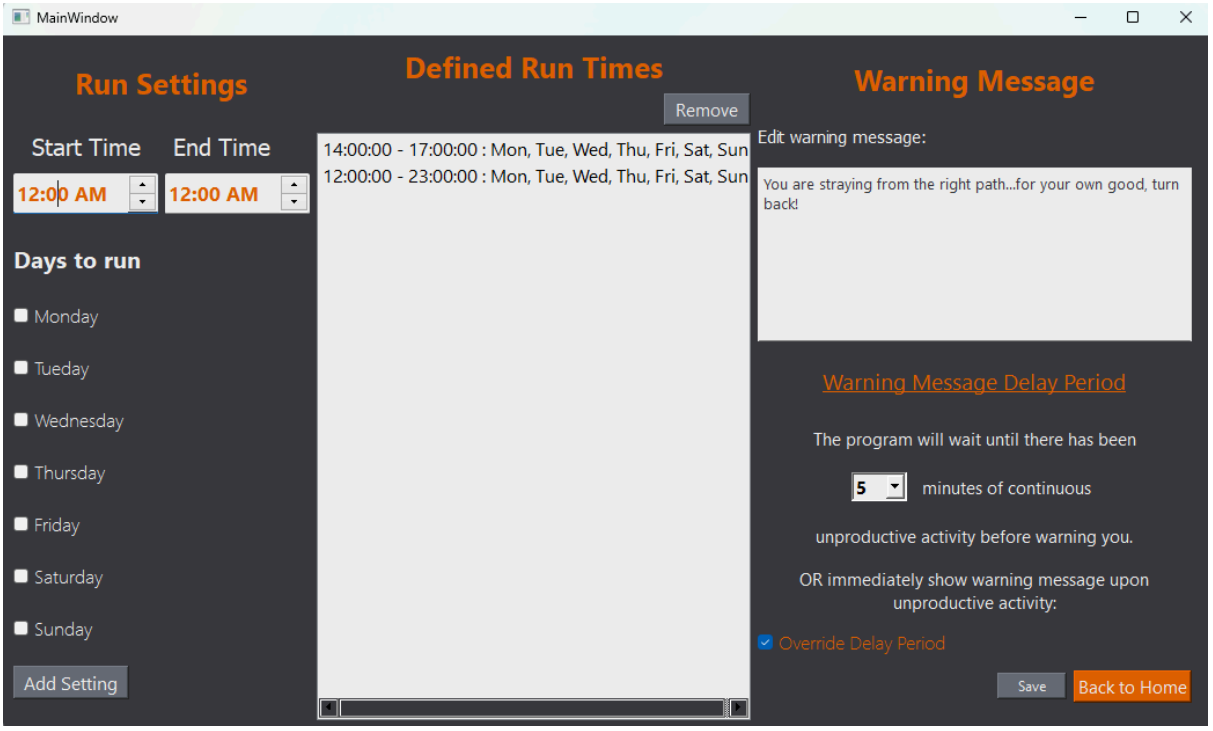
unproductive activity is being continuously accessed. The timer is used to measure time to know when to show the message in order to suit the chosen delay period. At any moment, if productive activity is detected, the timer is cancelled and the unproductive activity flag is set to False. This allows flexibility in how warnings are displayed, based on user preference.

Meaningful Variables

```
# Save the current settings back to the settings file
def save_settings(self):
    wait_time = self.ui.Delay.currentText()  # Get the selected wait time
    self.parent().set_wait_time(wait_time)  # Update the main window's wait time
    self.parent().sessions = self.sessions  # Update the main window's sessions
    self.parent().settings["sessions"] = self.sessions  # Update settings
    self.parent().settings["override_delay"] = self.OverrideDelay.isChecked()  # Update override delay
    self.parent().settings["warning_message"] = self.WarningMessage.toPlainText()  # Update warning message
    save_settings(self.parent().settings)  # Save the settings to the file
    print("Settings saved")
    self.reload_settings_page()  # Reload the settings page
```

```
self.unproductive_flag = False  # Flag indicating if unproductive activity is detected
self.unproductive_timer = QElapsedTimer()  # Timer to track unproductive activity duration
self.warning_displayed = False  # Whether a warning message is currently displayed
```

SettingsPage GUI



Configuration File


```
{
    "wait_time": "5",
    "sessions": [
        {
            "start_time": "10:00",
            "end_time": "17:00",
            "days": [
                true,
                true,
                true,
                true,
                true,
                true,
                true
            ]
        }
    ],
    "categories": {
        "Social Media": false,
        "Games": false,
        "Video Entertainment": false,
        "Visual Entertainment": true,
        "Forums": true,
        "Online Shopping": true,
        "Productivity Killers": true
    },
    "whitelisted_sites": [],
    "blacklisted_sites": [],
    "productive_sites": [
        "docs.google.com",
        "www.workramp.com/blog/what-is-leadership-development/",
        "www.workramp.com/blog/what-is-sales-training/",
        "slides.google.com",
        "mail.google.com",
        "scholar.google.com",
        "khanacademy.org",
        "coursera.org",
        "edx.org",
        "canva.com",
        "drive.google.com"
    ],
    "override_delay": false,
    "warning_message": "You are straying towards the dark side... For your own good, please return to being productive!"
}
```

```python
# Path to the settings file where user preferences are saved
SETTINGS_FILE = 'settings.json'

# Function to load settings from the settings file when the program starts
def load_settings():
    try:
        # Open the settings file and load the content as a dictionary
        with open(SETTINGS_FILE, 'r') as f:
            settings = json.load(f)
            # Convert session times from strings to QTime objects for easier manipulation
            for session in settings.get('sessions', []):
                if isinstance(session["start_time"], str):
                    session["start_time"] = QTime.fromString(session["start_time"], "HH:mm")
                if isinstance(session["end_time"], str):
                    session["end_time"] = QTime.fromString(session["end_time"], "HH:mm")
            return settings  # Return the loaded settings
    except FileNotFoundError:
        # If the settings file doesn't exist, return default settings
        return {
            "wait_time": "5",
            "sessions": [],
            "categories": {},
            "whitelisted_sites": [],
            "blacklisted_sites": [],
            "productive_sites": [],  # List of productive sites
            "override_delay": False,
            "warning_message": "Unproductive activity detected! For your own good, please return to being productive!"
```

Data Structures

```python
# Unproductive category names with example site links
self.category_sites = {
    "Social Media": ["facebook.com", "twitter.com", "instagram.com", "discord.com"],
    "Games": ["crazygames.com", "store.epicgames.com", "store.steampowered.com", "poki.com", "minesweeper.online"],
    "Video Entertainment": ["youtube.com", "netflix.com", "primevideo.com", "disneyplus.com", "sky.com"],
    "Visual Entertainment": ["globalcomix.com", "mangadex.org", "manganato.com", "anime-planet.com", "readallcomics.com"],
    "Forums": ["reddit.com", "quora.com", "discord.com", "steam.com"],
    "Online Shopping": ["amazon.com", "alibaba.com", "aliexpress.com", "zalora.com", "eBay.com", "taobao.com", "hktvmall.com"],
    "Productivity Killers": ["nytimes.com/games/wordle", "nytimes.com/puzzles/spelling-bee", "nytimes.com/crosswords/game/mini", ""]
}
```

```python
# Create a copy of the settings to save, converting QTime objects back to strings
settings_copy = {
    "wait_time": settings["wait_time"],
    "sessions": [
        {
            "start_time": session["start_time"].toString("HH:mm"),
            "end_time": session["end_time"].toString("HH:mm"),
            "days": session["days"]
        }
        for session in settings["sessions"]
    ],
```

File Input and Output

```
# Function to load settings from the settings file when the program starts
def load_settings():
    try:
        # Open the settings file and load the content as a dictionary
        with open(SETTINGS_FILE, 'r') as f:
            settings = json.load(f)
            # Convert session times from strings to QTime objects for easier manipulation
            for session in settings.get('sessions', []):
                if isinstance(session["start_time"], str):
                    session["start_time"] = QTime.fromString(session["start_time"], "HH:mm")
                if isinstance(session["end_time"], str):
                    session["end_time"] = QTime.fromString(session["end_time"], "HH:mm")
            return settings  # Return the loaded settings
    except FileNotFoundError:
        # If the settings file doesn't exist, return default settings
        return {
            "wait_time": "5",
            "sessions": [],
            "categories": {},
            "whitelisted_sites": [],
            "blacklisted_sites": [],
            "productive_sites": [],  # List of productive sites
            "override_delay": False,
            "warning_message": "Unproductive activity detected! For your own good, please return to being productive!"
        }
```

The function load_settings reads the stored settings in settings.json.

```
# Save the current settings back to the settings.json file
def save_settings(settings):
    # Create a copy of the settings to save, converting QTime objects back to strings
    settings_copy = {
        "wait_time": settings["wait_time"],
        "sessions": [
            {
                "start_time": session["start_time"].toString("HH:mm"),
                "end_time": session["end_time"].toString("HH:mm"),
                "days": session["days"]
            }
            for session in settings["sessions"]
        ],
        "categories": settings["categories"],
        "whitelisted_sites": settings["whitelisted_sites"],
        "blacklisted_sites": settings["blacklisted_sites"],
        "productive_sites": settings.get("productive_sites", []),  # Save productive sites if available
        "override_delay": settings.get("override_delay", False),
        "warning_message": settings.get("warning_message", "Unproductive activity detected!")
    }
    # Write the settings copy to the settings.json file
    with open(SETTINGS_FILE, 'w') as f:
        json.dump(settings_copy, f, indent=4)
```

The function save_settings writes the current settings back to the file, ensuring updates to user settings are stored, helping me meet my success criteria of saving user changes and settings. Load_settings and save_settings allows the user preferences to persist, enhancing the user experience.

Bibliography

Elan Maulani, I., Azis, I., Cahya, M. N., Komarudin, K., & Sagita, A. B. (2024). Implementation of object-oriented programming with Pyqt: Development of calculation application. Devotion : Journal of Research and Community Service, 5(1), 156-163. https://doi.org/10.59188/devotion.v5i1.679

Galarnyk, M. (2022, April 27). Logistic regression using Python (scikit-learn). Medium. https://towardsdatascience.com/logistic-regression-using-python-sklearn-numpy-mnist-handwriting-recognition-matplotlib-a6b31e2b166a

High DPI scaling in PyQt5. (2019, May 2). LeoMoon Studios -. https://leomoon.com/journal/python/high-dpi-scaling-in-pyqt5/

How to extract data using tesseract OCR? (2024, September 19). Docsumo - Document AI Platform Built for Scale & Efficiency. https://www.docsumo.com/blog/tesseract-ocr

Machine learning - Logistic regression. (n.d.). W3Schools Online Web Tutorials. https://www.w3schools.com/Python/python_ml_logistic_regression.asp

Python, R. (2021, February 3). QT designer and Python: Build your GUI applications faster – Real Python. Python Tutorials – Real Python. https://realpython.com/qt-designer-python/

Reading text from the image using tesseract. (2022, December 1). GeeksforGeeks. https://www.geeksforgeeks.org/reading-text-from-the-image-using-tesseract/

Sklearn.linear_model.LogisticRegression. (2023). scikit-learn. Retrieved September 20, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

TfidfVectorizer. (2023). scikit-learn. Retrieved September 20, 2024, from https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html