Technologic Papers

Design Patterns - איך לחשוב עם הקופסה

מאת עידו טולדנו ואור פרגר

הגלגל

מה הוא לא יכול לעשות? לגרום לחפצים דוממים לנוע, להפיק חשמל, להכשיל אנשים בבגרות במתמטיקה... כל אלו הנם רק חלק מהסיבות שהגלגל נחשב לאחת מההמצאות המכוננות של האנושות, כזו שאף אחד לא מנסה להחליף או להמציא מחדש בשל היעילות והאפקטיביות שלה בשלל השימושים שיש לה.

לא בכדי הומצא גם הביטוי "להמציא את הגלגל מחדש", אשר מתייחס לביצוע עבודה מיותרת ביצירת דבר אשר כבר קיים, וככל הנראה בצורה טובה יותר. בפיתוח תוכנה למשל, הביטוי יכול להתייחס למימוש קוד כלשהו שמבצע פונקציונליות שכבר סופקה ע"י שלל ספריות צד שלישי. אם, לדוגמה, נרצה להצפין מידע מסוים, כנראה שעדיף להשתמש בספרייה קיימת המממשת מודלי צופן בצורה טובה ומוצלחת מאשר לממש את את אותו מודל צופן מאפס. אם היינו מנסים לממש מודל צופן בעצמנו, סביר להניח שהתהליך היה אורך זמן רב שלא לצורך, ויתרה מכך, קיים סיכוי סביר שהיינו עושים טעות בדרך, ובכך מחלישים את מודל הצופן שכתבנו.

זוהי רק דוגמה אחת להימנעות ה"המצאת הגלגל מחדש", אך ישנם עוד מקרים רבים. במאמר זה נעסוק בבעיה זו בהיבט של פיתוח התוכנה, ובפיתרון שלה - Design Patterns.

אז מה זה Design Patterns ולמה זה טוב?

מראשיתו של עולם פיתוח התוכנה ועד היום (תקופה לא קצרה של כ-80 שנה), ניתן להגיד שפותחו לא מעט תוכנות. חלקן גדולות, חלקן קטנות, חלקן מקומפלות וחלקן אף צרובות. מאחר וכל כך הרבה תוכנות נכתבו עד היום, הגיוני שמפתחים רבים לאורך השנים נתקלו בבעיות שכבר התקילו מפתחים אחרים בעבר ובתקווה גם נפתרו על ידיהם.

כך חשבו גם אריך, ריצ'רד, ראלף וג'ון, הידועים בכינוי "כנופיית הארבעה" (gang of four), אשר טבעו את הביטוי design patterns (או בעברית - תבניות עיצוב) בספרם בעל השם ההולם: "design patterns". (לא להתבלבל עם "כנופיית הארבע" המקורית אשר ידועים בשל סיבות אחרות)

לפי "כנופיית הארבעה", Design Pattern היא "תיאור של תקשורת בין אובייקטים ומחלקות אשר עוצבו כדי לפתור בעיית עיצוב מסויימת בהקשר מסויים". אם להציע ניסוח פשוט יותר, design pattern היא מבנה פיתרון כללי לבעיה כללית כלשהי בפיתוח תוכנה.

אם נעזר ב"כנופיית הארבעה", ניתן לתאר design pattern באמצעות החלקים האלו:

- **מטרה:** מה ה-design pattern פותרת / משפרת? מה התכלית שלה?
- **מניע:** תרחיש המדגים את הבעיה שה-design pattern פותר ומתאר כיצד ה-design pattern עוזרת לפתור אותה.
- **יישומיות:** תיאור של המצבים בהם אותה design pattern תהיה שימושית.
- **מבנה:** תיאור של החלקים השונים שמרכיבים מימוש של ה-design pattern (ממשקים, ירושות וכדומה...).
- **השלכות:** כיצד ה-design patterns משיגה את מטרתה? מה הם היתרונות והחסרונות של השימוש בה?

אנו כמפתחים מרוויחים מספר לא מבוטל של יתרונות משימוש ב-design patterns:

ראשית, כפי שהוזכר בתחילת המאמר, ע"י שימוש ב-design patterns אנו לא משקיעים מאמצים מיותרים ב"המצאת הגלגל מחדש". שימוש ב-design pattern מאפשר לנו להשתמש בקונספט מוכר ומוכח על מנת לפתור בעיה, במקום להמציא פתרון חדש בעצמנו. ייתכן שנוכל למצוא פיתרון טוב בעצמנו, אך הדבר כנראה ייקח זמן רב יותר משמעותית, מה גם שאין הבטחה שאכן נוכל למצוא פיתרון טוב יותר (או פיתרון שלא נמצא קודם).

בנוסף לכך, הבנת נושא ה-design patterns מאפשר לנו להבין טוב יותר את התוכנות שעושות בהן שימוש. במקום להסתכל על תוכנה כלשהי כאוסף של מחלקות ופונקציות ייחודיות לתוכנה שלנו, design patterns מאפשרות לנו לאפיין ולהגדיר חלקים שונים של התוכנה באופן יותר מופשט ומאורגן. באמצעות הפשטה זו, אנו מסוגלים להבין תוכנה גם מבלי שנדע לעומק כיצד היא ממומשת (את נקודה זו נמחיש בשלב הדגמת שימוש ב-design pattern).

יתרון נוסף של שימוש ב-design pattern הינו הקטנת זמן הפיתוח ונוחות הרחבת התוכנה. כאשר אנו כותבים תוכנה שאנשים אחרים יעשו בה שימוש, סביר מאוד להניח שעם הזמן היא תעבור שינויים (תיקוני באגים, הרחבת פונקציונאליות וכו...). design patterns טובים בנויים כך שהרחבת והוספת תכולות ו/או שינוי תכונות של התכולות הקיימות לא יגררו שינוי רב בקוד, מאחר וניתן יהיה לעשות את השינויים במספר מצומצם כמה שניתן של מקומות בקוד. (על מנת להבין נקודה זו יותר לעומק, אתם מוזמנים לקרוא על עקרונות פיתוח התוכנה S.O.L.I.D)

עם כל זאת, חשוב להבין ש-design patterns הן אינן פתרון מדויק או מימוש של פתרון. לא ניתן להעתיק ולהדביק design pattern כמו שניתן לעשות לקוד. שימוש ב-design patterns דורש מחשבה ותכנון כיצד

יש ליישם אותן, על מנת שהיישום יהיה רלוונטי לבעיה אותה אנו פותרים ועל מנת שנמנע משימוש מופרז ב-design pattern. האם האמצעים העומדים לרשותנו מתאימים למימוש של התבנית (לדוגמה, ישנן design patterns אשר נוחות יותר למימוש בשפות מסויימות על פני אחרות)? האם התבנית רלוונטית לבעיה עמה אנו מתמודדים?

לדוגמה, כנראה שלא נצטרך להשתמש ב-design pattern כלשהי על מנת להדפיס "Hello World" יותר ביעילות (למרות שזה אפשרי, קישור לדוגמה יצורף בסוף המאמר), מאחר וזה כנראה יסבך את הקוד במידה שלא מצדיקה זאת. בנוסף, לא נרצה לתכנן ולממש design pattern בתוכנה שלנו ורק באמצע העבודה (במקרה הטוב) לגלות שה-design pattern כלל אינה רלוונטית לבעיה אותה אנו מנסים לפתור.

קצת דוגמאות

קיים מגוון גדול של design patterns מתועדות (וכנראה מגוון גדול אף יותר של design patterns לא מתועדות), אך היום אנו נתמקד בתבנית אחת: State.

מטרה?

State הנה design pattern אשר מאפשרת לאובייקט לשנות את התנהגותו לפי המצב (State) בו הוא נמצא בכל רגע נתון.

מניע (הבעיה)

כיצד ניתן ליצור אובייקט כך שיהיו לו מצבים שונים שעל פיהם תיקבע התנהגותו (שהוא יוכל להחליף בינהם), מבלי לסבך ולנפח יתר על המידה את המימוש של האובייקט עצמו? במילים אחרות, איך נאפשר לאובייקט להתנהג במספר אופנים שונים תוך כדי שאנחנו שומרים על המימוש של האובייקט עצמו פשוט ככל הניתן?

פתרון

פירוק המחלקה המייצגת את האובייקט למחלקות State אשר מייצגות את המצבים השונים בהם האובייקט יכול להיות ומימוש ההתנהגות האופיינית ל-State במחלקת ה-State.

דוגמה מעשית לבעיה ופתרון

מזל טוב! נבחרתם ע"י המנהל של Digital Whisper לממש אתר חדש המארח בלוגים בנושא דליפות מידע וריגול תעשייתי: Digital careless whisper! אחרי שניקיתם את הקונפטי ונפרדתם מהחקיין של ג'ורג' מייקל ששכרתם לכבוד אירוע ההכרזה, ניגשתם לעבודה.

טוב ויפה, האתר מתקדם ומקבל צורה, ואז אתם מתחילים לעבוד על הקוד של פרסום המאמרים. החלטתם שכל מאמר ייוצג ע"י אובייקט בשם "Article", אשר מתואר ע"י הממשק הבא:

מאפיינים:

1. state: מתאר את מצב המאמר, יכול להיות Draft (טיוטה, לא מוכן לפרסום), Moderated Draft (טיוטה סופית, מוכן לפרסום) או Published (פורסם).
2. name: שם המאמר.
3. publisher: כותב המאמר.
4. text: תוכן המאמר

מתודות:

1. add_text: מתודה העורכת את טקסט המאמר בהתחשב ב-state שלו, כך:
    1. מאמר במצב Draft: המתודה תאפשר להוסיף טקסט לתוכן הקיים.
    2. מאמר במצב Moderated Draft: המתודה תחזיר שגיאה מאחר ולא נרצה להוסיף טקסט למאמר בלי שהוא יעבור בדיקה.
    3. מאמר במצב Published: המתודה תחזיר שגיאה מאחר והמאמר כבר פורסם, לכן לא ניתן לשנותו.
2. ask_review: מתודה השולחת את המאמר לאישור בהתחשב ב-State שלו, כך:
    1. מאמר במצב Draft: המתודה תעביר את האובייקט למצב Moderated Draft.
    2. מאמר במצב Moderated Draft: המתודה לא תבצע דבר, שכן המאמר כבר במצב Moderated Draft.
    3. מאמר במצב Published: המתודה תחזיר שגיאה, שכן המאמר כבר פורסם ולא ניתן לבדוק אותו.
3. publish: מתודה המבצעת פרסום של המאמר בהתחשב ב-State שלו, כך:
    1. מאמר במצב Draft: המתודה תחזיר שגיאה, שכן על המאמר להיות מאושר לאחר בדיקה קודם לכן.
    2. מאמר במצב Moderated Draft: המתודה תפרסם את המאמר רק אם האישור ינתן על ידי אדמין, ותעביר את האובייקט למצב Published.
    3. מאמר במצב Published: המתודה תחזיר שגיאה, מאחר שהאתר לא מעודד spamming.
4. render: מתודה המבצעת render (כלומר תציג אותו למשתמש) למאמר בהתחשב ב-State שלו, כך:
    1. מאמר במצב Draft: המתודה לא תעשה render למאמר.
    2. מאמר במצב Moderated Draft: המתודה לא תעשה render למאמר.
    3. מאמר במצב Published: המתודה תעשה render למאמר.
5. change_state: מתודה אשר מקבלת state כקלט ומשנה את ה-state של ה-Article אליו. אין בהכרח צורך לייצא את פונקציונאליות זו למתודה משלה, אך דרך זו מאפשרת לנו לרכז את כל הפעולות שקורות בעת שינוי state (כמו בדיקת תקינות קלט או תיעוד באמצעות log-ים) למתודה אחת.

בנוסף נרצה לממש שני Exception-ים:

1. LogicError - שגיאה לוגית אשר תיזרק כאשר נעשה ניסיון לבצע פעולה לא הגיונית (כמו פרסום מאמר שכבר פורסם).
2. AuthorizationError - שגיאה אשר נזרקרת כאשר משתמש מנסה לבצע פעולה שאסור לו לבצע.

(מאחר ומטרת ההדגמה היא לממש שימוש ב-design pattern, לא התייחסנו לחלקים חשובים כמו בדיקת קלט וטיפול בשגיאות על מנת לא להסיח את הדעת מהרעיון המרכזי)

פסודו-קוד של הרעיון הנ"ל יראה כך:

```
CLASS Article
    ATTRIBUTE state
    ATTRIBUTE name
    ATTRIBUTE publisher
    ATTRIBUTE text

    METHOD add_text
        IF current state is Draft THEN
            APPEND given text to current text
        ELSE
            RETURN bad usage error
        ENDIF

    METHOD ask_review
        IF current state is Draft THEN
            SET current state to Moderated Draft
        ELSE
            RETURN bad usage error
        ENDIF

    METHOD publish
        IF current state is Moderated Draft THEN
            SET current state to Published
        ELSE
            RETURN bad usage error
        ENDIF

    METHOD render
        IF current state is Published THEN
            RETURN the article rendered
        ELSE
            RETURN value indicating article shouldn't be rendered
        ENDIF
```

מימוש של רעיון זה ייראה ב-python (השפה בה בחרתם לכתוב את האתר) כך:

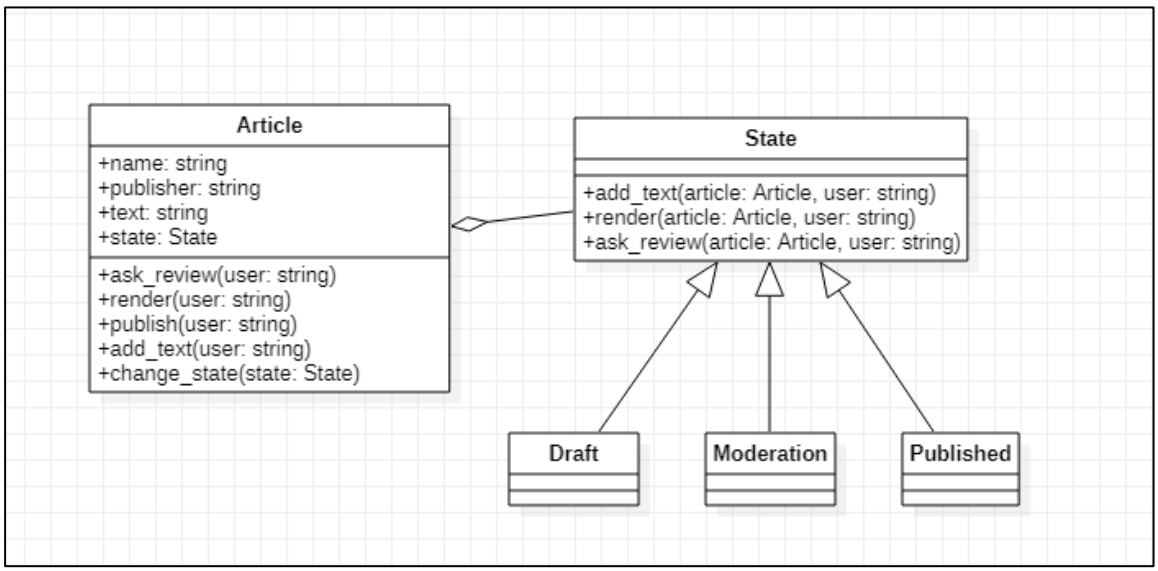
```
Class Article:
   def __init__(self, name, publisher):
      self.name = name
      self.publisher = publisher
      self.state = "Draft"
      self.text: str = ""

   def add_text(self, text):
      if self.state == "Draft":
         self.text += text
      else:
         raise Exception("Cannot add text if not a draft")

   def ask_review(self):
      if self.state == "Draft":
         self.state = "Moderation Draft"
      else:
         raise Exception("Cannot ask for review if not a draft")

   def publish(self):
      if self.state == "Moderation Draft":
         self.state = "Published"
      else:
         raise Exception("Cannot publish article if it hasn't been reviewed")

   def render(self):
      if self.state == "Draft":
         return ""  # A draft should not be rendered
      else if self.state == "Moderation Draft":
         return ""  # A moderation draft also should not be rendered
      else:
         return f"{self.name} (by: {self.publish}\n{self.text}"
```

הרעיון נחמד ובסה"כ עובד, אבל ישנה בעיה עם המימוש של המתודה publish: נכון לעכשיו ישנם רק שלושה מצבים אפשריים ל-article והפונקציונליות לכל אחד יחסית פשוטה, אך מה ייקרה אם נרצה להוסיף מצב חדש עם פונקצונליות ייחודית (ואולי מורכבת יותר)? מה אם במצב החדש הזה, add_text, render, publish ו-render יתנהגו כולן באופן שונה מהמתואר עד כה? נצטרך לערוך את כל הפונקציות הרלוונטיות בהתאם. תכנית זו אולי תעבוד טוב בטווח הקצר, אך ככל שנוסיף יותר state-ים כך תהפוך המחלקה article למורכבת יותר ובאופן כללי הקוד שלנו יהפוך לפחות קריא ופחות נוח לתחזוק באופן משמעותי.

אך אל דאגה, בדיוק כאן ה-State design pattern נחלץ לעזרתנו:

במקום שהמאפיין state יהיה מחרוזת כלשהי, ניצור מחלקה אבסטרקטית State, אשר מתארת כיצד state אמור להיראות ולהתנהג, ומחלקות state-ים שיורשות ממנה וממשות את הפונקציות הרלוונטיות להן בעצמן (לפי מה שמתואר במחלקה state), בעת הצורך.

המבנה של התוכנה בשימוש עם state-ים ייראה כך:



ועכשיו להדגמה

כעת נדגים מימוש מחדש של האובייקט "Article" באמצעות שימוש ב-state design pattern.

כפי שמתואר במבנה התוכנה, נרצה לכתוב מחלקה בשם State אשר מגדירה את הממשק שכל state של article צריך לממש:

```
class AuthorizationError(Exception):
    """ Authorization error, thrown when a user tries to do something he is not
allowed to """
    pass


class LogicError(Exception):
    """ Logic error, thrown when someone isn't following the correct steps of
publishing an article """
    pass


class State:
    """ A representation of the article's state """

    def add_text(self, article, text):
        """ Add text to the article according to it's state """
        raise LogicError("Can't add text in the current state")

    def render(self, article, user):
        """ Render the article according to it's state and the requesting user
"""
        raise LogicError("Can't render article in the current state")

    def ask_review(self, article, user):
        """ Ask for review on the article according to it's state and the
requesting user """
        raise LogicError("Can't ask review in the current state")

    def publish(self, article, user):
        """ Publish the article according to it's state and the requesting user
"""
        raise LogicError("Can't publish article in the current state")
```

יש לציין שהמחלקה State עצמה **אינה** מממשת אף פונקציה (אלא זורקת שגיאה במקום), מאחר והיא אינה state בעצמה אלא רק "תבנית" של state.

כעת שיש לנו ממשק לאיך מחלקת "state" צריכה להראות, נרצה לממש את שלושת ה-state-ים של ה-article (draft, moderation, publlished). נעשה זאת ע"י יצירת 3 מחלקות, כאשר כל אחת מהן יורשת מהמחלקה State וממשת את המתודות המוגדרות בה לפי צורכן. לכל state מימוש שונה למתודות (לפי איך שתיארנו בממשק של המחלקה "Article".), אך ה-signature של כל מתודה יהיה זהה לזו של State:

```
class Draft(State):

    def add_text(self, article, text):
        """ Add text to the article """
        article.text += text

    def render(self, article, user):
        """ Render the article only if the user requesting it is the publisher
"""
        if user != article.publisher:
            raise AuthorizationError("Not publisher")

        return article.text

    def ask_review(self, article, user):
        """ Ask for review only if the user requesting it is the publisher """
        if user != article.publisher:
            raise AuthorizationError("Not publisher")

        article.change_state(Moderation)


class Moderation(State):
    def render(self, article, user):
        """ Render the article only if the user requesting it is the publisher
or the site admin """
        if user != article.publisher and user != "admin":
            raise AuthorizationError("Not user or admin")

        return article.text

    def publish(self, article, user):
        """ Publish the article according to it's state """
        if user != "admin":
            raise AuthorizationError("Not admin")

        article.change_state(Published)


class Published(State):
    def render(self, article, user):
        """ Render the article to whomever (NOT whoever) wishes to see it """
        return f"PRESENTING: {article.name}\n" \
               f"BY: {article.publisher}\n" \
               f"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n" \
               f"{article.text}\n" \
               f"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"
```

השלכות

עתה שיש לנו state-ים ל-article אשר מממשים כיצד article צריכה להתנהג במצביה השונים, נוכל סוף סוף לממש מחלקה article אשר מתנהגת בצורה ייחודית בהתאם לכל state שהיא נמצאת בו ביתר קלות:

```
class Article:
    """ An article representation """

    def __init__(self, name, publisher):
        self.name = name
        self.publisher = publisher
        self.state = Draft()
        self.text = ""

    def ask_review(self, user):
        """ Ask for review of the article """
        return self.state.ask_review(self, user)

    def render(self, user):
        """ Render the article """
        return self.state.render(self, user)

    def publish(self, user):
        """ Publish the article """
        self.state.publish(self, user)

    def add_text(self, text):
        """ Add text to the article """
        self.state.add_text(self, text)

    def change_state(self, state):
        """ Change the state of the article """
        self.state = state()
```

כאן בא לידי ביטוי היתרון הגדול של ה-state design pattern. אם היינו מממשים את article לפי העיצוב הראשוני, היינו צריכים לבצע בדיקות לגבי המצב של המאמר ולפיו לבצע התנהגות מסויימת. לעומת זאת, ייצוא מימוש הפונקציונליות של כל state למחלקה נפרדת מאפשרת לנו להפוך את המימוש של התנהגות ה-article לפשוטה עד לכדי שורת קוד אחת!

בנוסף באופן זה נוכל לשנות את המימוש של ה-state-ים השונים ואף להוסיף state-ים חדשים מבלי לשנות דבר במחלקה article.

כעת ניתן ליצור ולפרסם מאמרים באופן הזה:

```
def main():
    publisher = "Ido & Or"
    # Create a new article and add text to it
    new_article = Article("Design Patterns Jokes", publisher)
    new_article.add_text("Q: Why was the interface sad?\n")
    new_article.add_text("A: It wasn't fulfilled")

    # Send it for review as the publishers
    print("-- Asking review --")
    new_article.ask_review(publisher)

    # Publish the article as the publisher
    print("-- Publishing as publisher --")
```

```
    try:
        new_article.publish(publisher)
    except AuthorizationError as e:
        print(f"Failed with exception: {e}")

    # Publish it as admin
    print("-- Publishing as admin --")
    new_article.publish("admin")

    # Read it as a guest user
    print("-- Rendering the article as a guest --")
    print(new_article.render("Guest"))

    # Publish it again and get an error
    print("-- Publishing the article once again --")
    try:
        new_article.publish("admin")
    except LogicError as e:
        print(f"Failed with exception: {e}")


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

והפלט שיתקבל יהיה:

```
-- Asking review --
-- Publishing as publisher --
Failed with exception: Not admin
-- Publishing as admin --
-- Rendering the article as a guest --
PRESENTING: OOP Jokes
BY: Ido & Or
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Q: Why was the interface sad?
A: It wasn't fulfilled
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- Publishing the article once again --
Failed with exception: Can't publish article in the current state

Process finished with exit code 0
```

יש לשים לב שבין אם היינו משתמשים ב-design pattern או לא, השימוש במחלקה article נותר זהה, אך כעת נוכל להוסיף state-ים חדשים מבלי לשנות את המחלקה article עצמה! (בהנחה שלא נרצה לשנות את הממשק של article)

חשוב גם להבחין שבעוד שאנחנו בחרנו לבחור במאמרים על מנת להמחיש את השימוש ב- state design pattern, התבנית מתאימה לכל מימוש של אובייקט אשר מתנהג בהתאם למצב בו הוא נמצא. היינו יכולים להשתמש ב-state design pattern גם על מנת לייצג מכונית אשר מתנהגת שונה בהתאם להילוך בו היא נמצאת. גם את המבנה של המאמר וגם את המבנה של המכונית היינו יכולים יותר לעומק, מאחר ואנחנו מכירים את ה-state design pattern ואת המבנה של אובייקטים בו. היינו יכולים אפילו להוסיף הילוך למכונית בקלות יותר ומבלי חשש להרוס את המימוש של המכונית או של שאר ההילוכים.

בנוסף, חשוב להבין שלא בהכרח היינו צריכים להשתמש ב-state design pattern במקרה הזה. אם ל-article היו רק 2 מצבים, או שה-state של article משתנה לעתים רחוקות, מימוש של ה- state design pattern היה יכול להיות פיתרון לסובך מדי ביחס לבעיה.

פיתרון דומה, שפה שונה

אחת הסיבות לכך שהקונספט של design patterns הוא כה משמעותי ושימושי היא האונבירסליות שלו. ה-design pattern שהצגנו בקטע הקודם, כמו גם design patterns רבות אחרות אינן מוגבלות לשפה מסויימת, חלק אינן מוגבלות אפילו לשפות high level (אך נושא זה חורג ממסגרת המאמר).

גם את עיקרון נדגים באמצעות ה-state design pattern. נוכל לקחת את ה-design והפעם לממש אותו בצורה דומה בשפה c++:

ראשית, עשינו include-ים למספר ספריות סטנדרטיות, הנחוצות לנו בתוכנית. לאחר מכן, ביצענו Forward Declaration למחלקות הרלוונטיות שמימושן יגיע בהמשך. לא נרחיב על כאן על הסיבה לצורך בטכניקה זו. בנוסף, הגדרנו exception ייעודי לבעיית הזדהות:

```
#include <iostream>
#include <string>
#include <exception>

// Forward declaration
class Article;
class Draft;
class Moderation;
class Published;

// Authentication error exception class
struct AuthenticationError : public std::exception {
    const char* what() const throw () {
        return "Authentication Error";
    }
};
```

```
// A representation of the article's state
class ArticleStateBase {
public:
    virtual ~ArticleStateBase() {}

    // Add text to the article according to it's state
    virtual void AddText(Article& article, std::string text)
    {
        throw std::logic_error("Can't add text in the current state");
    }
```

```
    // Render the article according to it's state and the requesting user
    virtual std::string Render(Article& article, std::string user)
    {
        throw std::logic_error("Can't render article in the current state");
    }

    // Ask for review on the article according to it's state and the requesting user
    virtual void AskReview(Article& article, std::string user)
    {
        throw std::logic_error("Can't ask review in the current state");
    }

    // Publish the article according to it's state and the requesting user
    virtual void Publish(Article& article, std::string user)
    {
        throw std::logic_error("Can't publish article in the current state");
    }
};
```

בדומה לקוד פייתון, הגדרנו למחלקה האבסטרקטית לזרוק שגיאה כמימוש דיפולטי לכל פונקציה רצויה. לאחר מכן, נגדיר את מחלקת Article כך שתקרא לפונקציות לפי ה-state הרלוונטי, ונגדיר את ה-states השונים:

```
// An article representation
class Article {
public:
    Article(std::string name, std::string publisher);

    // Add text to the article according to it's state
    void AddText(std::string text)
    {
        m_state->AddText(*this, text);
    }

    // Render the article according to it's state and the requesting user
    std::string Render(std::string user)
    {
        return m_state->Render(*this, user);
    }

    // Ask for review on the article according to it's state and the requesting user
    void AskReview(std::string user)
    {
        m_state->AskReview(*this, user);
    }
```

```
    // Publish the article according to it's state and the requesting user
    void Publish(std::string user)
    {
        m_state->Publish(*this, user);
    }

    void ChangeState(ArticleStateBase* new_state) {
        delete m_state;
        m_state = new_state;
    }

    std::string m_text;
    std::string m_name;
    std::string m_publisher;
    ArticleStateBase* m_state;
};
```

```
class Published : public ArticleStateBase {
public:
    virtual ~Published() {}

    // Render the article to whomever (NOT whoever) wishes to see it
    virtual std::string Render(Article& article, std::string user) override
    {
        return std::string("PRESENTING: ") + article.m_name +
            std::string("\nBY: ") + article.m_publisher +
            std::string("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n") +
            article.m_text +
            std::string("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    }
};
```

```
class Moderation : public ArticleStateBase {
public:
    virtual ~Moderation() {}

    // Render the article only if the user requesting it is the publisher or the site admin
    virtual std::string Render(Article& article, std::string user) override
    {
        if (article.m_publisher.compare(user) != 0 || user.compare("admin") != 0) {
            throw AuthenticationError();
        }

        return article.m_text;
    }

    // Publish the article according to it's state
```

```
    virtual void Publish(Article& article, std::string user) override
    {
        if (user.compare("admin") != 0) {
            throw AuthenticationError();
        }

        article.ChangeState(new Published());
    }
};
```

```
class Draft : public ArticleStateBase {
public:
    virtual ~Draft() {}

    // Add text to the article
    virtual void AddText(Article& article, std::string text) override
    {
        article.m_text += text;
    }

    // Render the article only if the user requesting it is the publisher
    virtual std::string Render(Article& article, std::string user) override
    {
        if (article.m_publisher.compare(user) != 0) {
            throw AuthenticationError();
        }

        return article.m_text;
    }

    // Ask for review only if the user requesting it is the publisher
    virtual void AskReview(Article& article, std::string user) override
    {
        if (article.m_publisher.compare(user) != 0) {
            throw AuthenticationError();
        }

        article.ChangeState(new Moderation());
    }
};
```

```
Article::Article(std::string name, std::string publisher) :
    m_text(), m_name(name), m_publisher(publisher), m_state(new Draft())
{}
```

כאן מימשנו את ה-constructor של מחלקת ה-Article שלנו. הסיבה לכך שמימוש ה-constructor מופרד משאר מחלקת ה-Article שלנו, היא שה constructor מאתחל את m_state על ידי יצירת Draft, ולשם כך נחוץ קודם כל שמחלקת Draft תהיה מוגדרת היטב.

ולבסוף, פונקציית ה-main של התוכנית שלנו:

```
int main()
{
    std::string publisher("Ido & Or");
    // Create a new article and add text to it
    Article new_article("Design Patterns Jokes", publisher);
    new_article.AddText("Q: Why was the interface sad?\n");
    new_article.AddText("A: It wasn't fulfilled");

    // Send it for review as the publishers
    std::cout << "-- Asking review --" << std::endl;
    new_article.AskReview(publisher);

    // Publish the article as the publisher
    std::cout << "-- Publishing as publisher --" << std::endl;
    try {
        new_article.Publish(publisher);
    }
    catch (AuthenticationError& e) {
        std::cout << "Failed with exception: " << e.what() << std::endl;
    }

    // Publish it as admin
    std::cout << "-- Publishing as admin --" << std::endl;
    new_article.Publish("admin");

    // Read it as a guest user
    std::cout << "-- Rendering the article as a guest --" << std::endl;
    std::cout << new_article.Render("Guest") << std::endl;

    // Publish it again and get an error
    std::cout << "-- Publishing the article once again --" << std::endl;
    try {
        new_article.Publish("admin");
    }
    catch (std::logic_error& e) {
        std::cout << "Failed with exception: " << e.what() << std::endl;
    }

    return 0;
}
```

פיתרון שונה, שפה שונה

נשים לב שהמימוש שהצגנו ב-c++ דומה מאוד למימוש בשפת פייתון, ונאלצנו לבצע שינויים מינוריים על מנת להתאים את ה-state pattern לשפת התכנות.

פלט התוכנית הינו זהה לפלט תוכנית ה-python. עם זאת, לעיתים ניתן להשתמש בפיצ'רים של השפה על מנת לכתוב את התוכנית שלנו במספר דרכים שונות. נדגים כעת תוכנית הכתובה ב-c++ (ומשתמשת בפיצ'רים של c++14 והספריה הסטנדרטית), אשר אינה מממשת את ה-state pattern הקלאסי, ויש לה יתרונות וחסרונות שונים ביחס לתוכנית שהצגנו קודם:

```
#include <iostream>
#include <string>
#include <exception>
#include <type_traits>

// Authentication error exception class
struct AuthenticationError : public std::exception {
    const char* what() const throw () {
        return "Authentication Error";
    }
};

// Definition of different states
class ArticleStateBase {};
class Draft : ArticleStateBase {};
class Moderated : ArticleStateBase {};
class Published : ArticleStateBase {};
```

תחילת הקוד דומה מאוד לקוד שכתבנו קודם. הפעם עשינו include נוסף לספריית type_traits, שמיד נראה את הצורך בה. בנוסף, הגדרנו כמחלקה ריקה את ArticleStateBase,, וכמחלקות ריקות את ה-states השונים של המאמר (לא להתבלבל עם ה-forward declaration שביצענו בתוכנית הקודמת!), כך שהם יורשים מ-ArticleStateBase. מיד נראה מה הסיבה להגדרת המחלקות הריקות הללו.

נגדיר כעת את מחלקת ה-Article שלנו:

```
// An article representation
template<class ArticleState = Draft>
class Article
{
    static_assert(std::is_base_of<ArticleStateBase, ArticleState>::value, "ArticleState
 must inherit from ArticleStateBase");
public:
    Article(std::string name, std::string publisher, std::string text = {}) :
        m_text(text), m_name(name), m_publisher(publisher)
    {}
```

```
private:
    std::string m_text;
    std::string m_name;
    std::string m_publisher;
```

הפעם, כאשר הגדרנו את מחלקת ה-Article שלנו, הגדרנו אותה כ-template-ית, כך שה- default template argument שלה הוא מחלקת ה-Draft - שזהו למעשה ה-state שבו המאמר צריך להתחיל. לאחר מכן, הכנסנו static assert (הכוונה היא לבדיקה שנקראת בזמן קומפילציה, ואם הבדיקה נכשלת, הקוד לא מתקמפל). הבדיקה שאנחנו מבצעים היא שה-state שקיבלנו כ-template argument למעשה יורש ממחלקת ArticleStateBase. לאחר מכן, הגדרנו את ה-constructor של המחלקה והשדות הפרטיים של המחלקה.

כעת אנו מגיעים לחלק המעניין - הפונקציות השונות התלויות ב-state בו אנחנו נמצאים. הפונקציות כמעט זהות לפונקציות מהתוכנית הקודמת, לכן לא נתעכב על כולן, וניתן דוגמת מימוש לאחת הפונקציות:

```
    // Add text to the article
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Draft>::value, void> AddText(std::string
text)
    {
        m_text += text;
    }
```

מה בעצם קורה כאן? הפונקציה שאנחנו מגדירים כאן הינה AddText, וניתן לראות שתוכן הפונקציה הינו מאוד פשוט, ודומה ל-AddText מהתוכנית הקודמת.

הפעם, המתודה AddText גם היא template-ית, והגדרנו שה-template argument שלה - T - יקבל את הערך של ArticleState (של מחלקת ה-Article עצמה). הסיבה לכך היא שאנחנו צריכים שיהיה קשר בין הפונקציה לבין ה-ArticleState שהקומפיילר יהיה מודע אליו, שיתבטא בשימוש ב-std::enable_if_t[1]

```
    template<class T = ArticleState>
```

מה בעצם מבצע קטע הקוד הבא?

```
    typename std::enable_if_t<std::is_same<T, Draft>::value, void>
```

std::enable_if_t לוקח שני template arguments: הראשון - ערך בוליאני שצריך להיות ידוע בזמן קומפילציה (true או false). השני - type כלשהו. אם הערך הבוליאני הינו true, הפונקציה AddText תיכנס למחלקה ותקומפל, אחרת לא. במידה וקיבלנו true, enable_if_t לוקח את ה-type שנתנו כאגרומנט שני, ומשים אותו בקוד כאילו היינו כותבים את ה-type ישירות.

הבדיקה שאנחנו מבצעים היא השוואה בין T, כלומר ArticleState, לבין המחלקה Draft - אם ה-ArticleState הוא Draft, הפונקציה AddText תהיה קיימת, ותיראה כך:

```
    // Add text to the article
    template<class T = ArticleState>
    void AddText(std::string text)
    {
        m_text += text;
    }
```

הפונקציה אומנם template-ית, אך אין שימוש אמיתי ל-T בפונקציה כעת, לאחר שדרשנו ש-T יהיה Draft באופן דומה, מוגדרות שאר הפונקציות של המחלקה:

```
    // Ask for review only if the user requesting it is the publisher
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Draft>::value, Article<Moderated>> AskRev
iew(std::string user)
    {
        if (m_publisher.compare(user) != 0) {
            throw AuthenticationError();
        }

        return Article<Moderated>(m_name, m_publisher, m_text);
    }

    // Publish the article according to it's state
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Moderated>::value, Article<Published>> Pu
blish(std::string user)
    {
        if (user.compare("admin") != 0) {
            throw AuthenticationError();
        }

        return Article<Published>(m_name, m_publisher, m_text);
    }
```

אלה הן מתודות המעבר שלנו. באופן כללי, צריך להיזהר עם החזרת אובייקטים ממתודות ב-c++, השימוש כאן נעשה בצורה כזאת שמובטח לנו שהקומפיילר יבצע copy elision[2], לכן אין בכך בעיה.

מה שנותר לנו הוא לממש את מתודות ה-Render:

```
    // Render the article only if the user requesting it is the publisher
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Draft>::value, std::string> Render(std::s
tring user)
    {
        if (m_publisher.compare(user) != 0)
```

```
        {
            throw AuthenticationError();
        }

        return m_text;
    }

    // Render the article only if the user requesting it is the publisher or the site a
dmin
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Moderated>::value, std::string> Render(st
d::string user)
    {
        if (m_publisher.compare(user) != 0 || user.compare("admin") != 0)
        {
            throw AuthenticationError();
        }

        return m_text;
    }

    // Render the article to whomever (NOT whoever) wishes to see it
    template<class T = ArticleState>
    typename std::enable_if_t<std::is_same<T, Published>::value, std::string> Render()
    {
        return std::string("PRESENTING: ") + m_name +
            std::string("\nBY: ") + m_publisher +
            std::string("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\n") +
            m_text +
            std::string("\n~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~");
    }
```

ובכן, איך כעת יראה שימוש במחלקה שלנו?

```
 int main()
{
    std::string publisher("Ido & Or");
    // Create a new article and add text to it
    Article<> draft("Design Patterns Jokes", publisher);
    draft.AddText("Q: Why was the interface sad?\n");
    draft.AddText("A: It wasn't fulfilled");

    // Send it for review as the publishers
    std::cout << "-- Asking review --" << std::endl;
    Article<Moderated> moderated = draft.AskReview(publisher);

    // Publish the article as the publisher
    std::cout << "-- Publishing as publisher --" << std::endl;
```

```
    try {
        moderated.Publish(publisher);
    }
    catch (AuthenticationError& e) {
        std::cout << "Failed with exception: " << e.what() << std::endl;
    }

    // Publish it as admin
    std::cout << "-- Publishing as admin --" << std::endl;
    Article<Published> published = moderated.Publish("admin");

    // Read it as any user
    std::cout << "-- Rendering the article as a guest --" << std::endl;
    std::cout << published.Render() << std::endl;

    return 0;
}
```

כמובן שגם הפעם, פלט התוכנית הינו זהה. אך מה הם היתרונות והחסרונות של דרך המימוש הזו?

יתרונות:

- אין צורך בפונקציות וירטואליות, ובקריאה לפונקציה לפי המצביע הנוכחי של ה-state. היתרון הוא ביצועים בזמן ריצה.
- לא ניתן, בזמן קומפילציה, לקרוא לפונקציות שלא קיימות ב-state המתאים, במקום בזמן ריצה. לדוגמה, נקבל שגיאת קומפילציה אם ננסה להריץ:

```
Article<> draft("Design Patterns Jokes", "Ido & Or");
draft.Publish("admin");
```

- שליטה מלאה יותר על חתימות המתודות בהתאם ל-state. לדוגמה, כאשר המאמר במצב Published, אין צורך להעביר כפרמטר את המשתמש, שכן אין בדיקה שלו בכל מקרה.

חסרונות:

- קוד פחות קריא
- יצירה של מספר משתני Article שונים - לדוגמה, גם לאחר הקריאה לפונקציה draft.AskReview, עדיין קיים המשתנה draft, והוא עדיין שמיש (לדוגמה, ניתן לקרוא לפונקציה AskReview עליו פעם נוספת). ניתן לפתור זאת על ידי הכנסתו למעין zombie state, לדוגמה על ידי שימוש בשדה בוליאני is_valid, בדיקתו לפני כל פונקציה, וזריקת שגיאה בהתאם. כמובן שפתרון זה פוגע ביתרון של ביצועים בזמן ריצה.
- יצירת Article שאינו Draft ישירות. ניתן לפתור בעיה זו על ידי שילוב של הטכניקה שהצגנו עם enable_if, יחד עם constructor פרטיים ושימוש ב-friend functions[3]. לא נרחיב על פתרון זה, והקורא המעוניין מוזמן לנסות לממש בעצמו פתרון זה.

- עדיין ניתן לקרוא, אם ממש רוצים, לכל מתודה של המחלקה מכל מצב. כשהגדרנו את T במתודות השונות, אמרנו לו לקבל באופן דיפולטי את ArticleState. אבל, ניתן למעשה לשנות את T, ולדוגמה לקרוא למתודה AddText כאשר האוביקט במצב Moderated:

```
Article<> draft("Design Patterns Jokes", "Ido & Or");
Article<Moderated> moderated = draft.AskReview("Ido & Or");
moderated.AddText<Draft>("Blah blah");
```

וכעת, תוספת הטקסט הזו תימצא גם באובייקט moderated.

סיכום

Design Patterns הוא קונספט חשוב בעולם פיתוח התוכנה. הוא מאפשר לנו לפתח תוכנה באופן מהיר ויעיל יותר על ידי שימוש בפתרונות שהוכיחו את עצמם בעבר. נוסף על כך, הוא מאפשר לנו לחשוב על תוכנה באופן יותר מאורגן, דבר המאפשר לנו הבנה יותר מעמיקה על איך שתוכנה עובדת ללא צורך לדעת כיצד היא ממומשת.

במאמר זה הצגנו את ההיסטוריה של design patterns. הסברנו מהן design patterns וכיצד ניתן להשתמש בהן כדי לשפר את איך שאנו מפתחים ומבינים תוכנה. כמו כן הסברנו את המגבלות הקיימות לשימוש ב-design patterns.

לאחר מכן הדגמנו שימוש מעשי ב-design pattern ע"י הדגמה של ה-state design pattern. תארנו איזו בעיה design pattern זו נועדה לפתור, כיצד היא עושה זאת וכיצד ניתן לממש אותה באופן תיאורטי (מבנה). לאחר מכן הצגנו מימוש של design pattern זו וראינו כי היא אינה ייחודית לשפה אחת, אלא ניתנת לשימוש בכמה שפות.

כאשר תכתבו קוד שסביר להניח שאנשים נוספים יעבדו עליו, בהווה או בעתיד, אנו ממליצים לכם לדבוק במימוש ה-state pattern הקלאסי - הוא יותר אוניברסלי, מתאים למימוש בכמעט כל שפת תכנות מודרנית, יותר ברור וקריא לאחרים וכו'. מצד שני, אם אתם נדרשים לכתוב קוד בעל ביצועים גבוהים, ותוכלו להבטיח שיבצעו שימוש נכון במחלקה שלכם, נמליץ לכם לממש את המחלקה שלכם כמו בדוגמה האחרונה.

מאמר זה מהווה הקדמה לקונספט של design patterns. קיימות design pattern נוספות רבות אשר נועדו לפתור סוגים שונים של בעיות. אנו מעודדים אתכם להמשיך ולקרוא על design patterns נוספות, ואף ליצור חדשות משלכן. סביר להניח שאתם משתמשים ב-design pattern מוכר (או לא) מבלי לדעת זאת, וגם אם לא סביר מאוד שישנה בעיה שנתקלתם בה (או תיתקלו בה) אשר יכולה להפתר ע"י design pattern זו או אחרת. כך או כך, אין ספק שהיכרות עם design pattern-ים יכולה להפיק לכם תועלת רבה.

קריאה נוספת

Design Patterns by Gang of four - הספר שנכתב ע"י מי שפיתחו את נושא ה-design patterns בו הם מסבירים לעומק על הקונספט של design pattrs וכן על מספר design patterns קלאסיים.

Refactoring guru - אתר מצויין המסביר על design patterns, עקרונות פיתוח SOLID ועוד...

The Abuse of Design Patterns in writing a Hello World Program - פוסט שמדגים כיצד ניתן להשתמש ב-design patterns באופן מופרז על ידי שימוש ב-4 design pattern-ים שונים!

ביבליוגרפיה

- Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
- https://refactoring.guru/
- https://en.wikipedia.org/wiki/Software_design_pattern
- https://stackoverflow.com/questions/6972368/stdenable-if-to-conditionally-compile-a-member-function

הערות שוליים

1. https://en.cppreference.com/w/cpp/types/enable_if
2. https://en.cppreference.com/w/cpp/language/copy_elision
3. https://en.cppreference.com/w/cpp/language/friend

על הכותבים

אור פרגר: אוהב ללמוד גרמנית ולשמוע ביקורות. דעות ושאלות במייל: ororp3@gmail.com.

עידו עידוט טולדנו: הסדרה האהובה היא How I Met Your Mother, צופיפניק לשעבר ואוהב לעזור. דעות ושאלות במייל: idotol123@gmail.com.