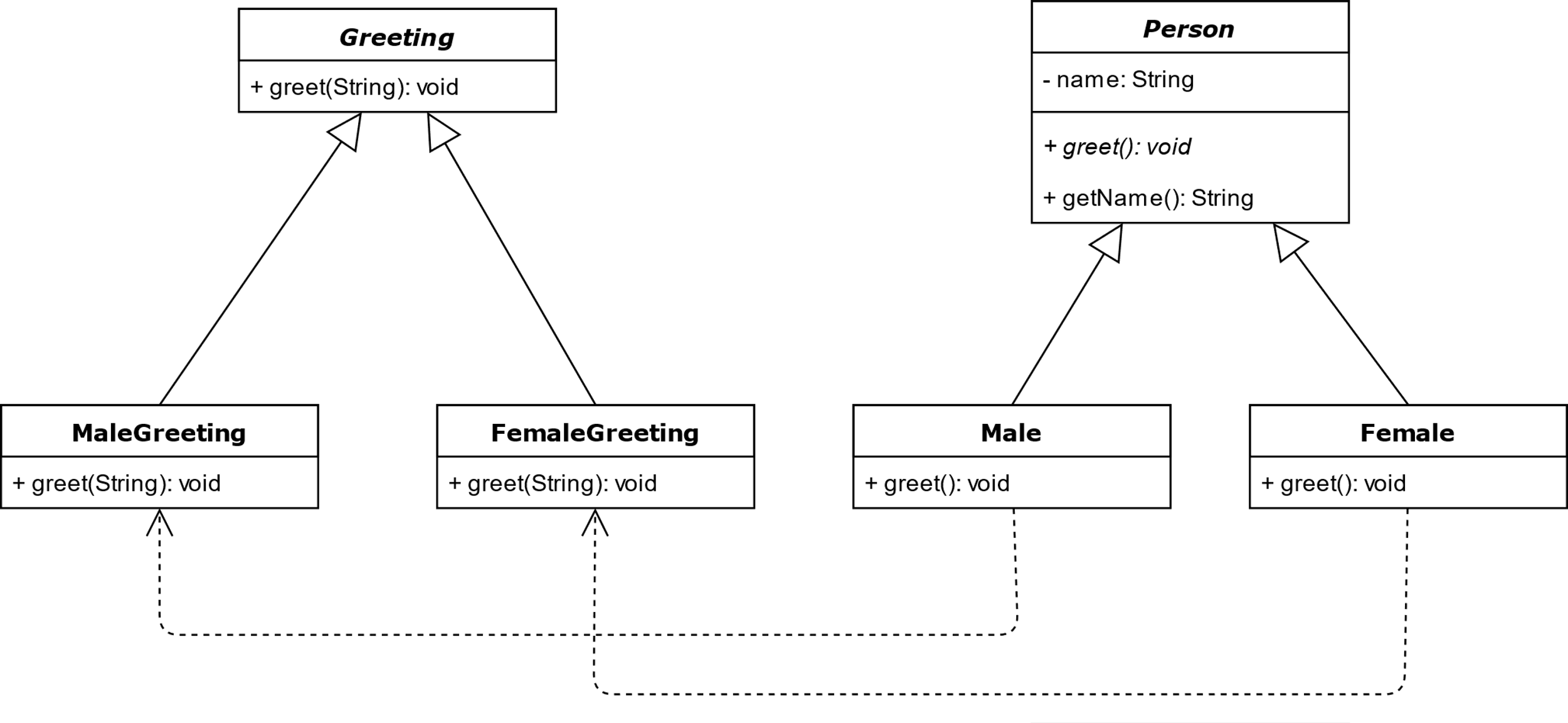
שאלה 1

1. פלט התוכנית:

Hello Mr. Danny

Hello Ms. Danna

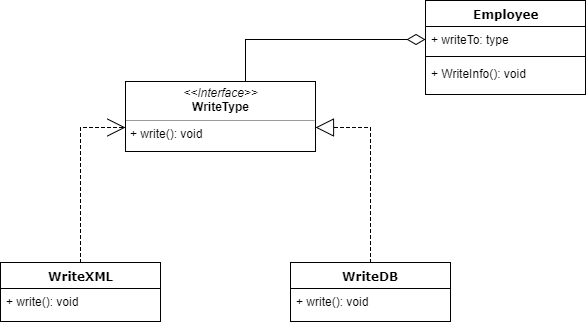
1. ממומש creational design pattern ע"י factory method (המתודה Person.greet()), בכך הוא פותר את הבעיה של יצירת המחלקה היורשת המתאימה של Greeting כאשר צריך ליצור אותה במתודה Person.greet().
2. Person.greet() הוא factory method, המחלקות Male ו-Female דורסות את המתודה greet() ובה כל אחת יוצרת את האובייקט המתאים של Greeting – MaleGreeting ו- FemaleGreeting בהתאמה ואז קוראות למתודה Greeting.greet(String) של אובייקט זה עם שם האדם (השדה name).



שאלה 2

1. הפתרון המוצע מפר את עיקרון הפתיחות/סגירות שכן במידה ונרצה להוסיף עוד יכולת כתיבה עתידית למשל לשרת SQL נצתרך להוסיף פונקציה חדשה, ולגעת בכל הקוד שקשור לקריאות פונצקיות אלה.

לעומת זאת במידה ונחליף את המימוש במימוש הבא:

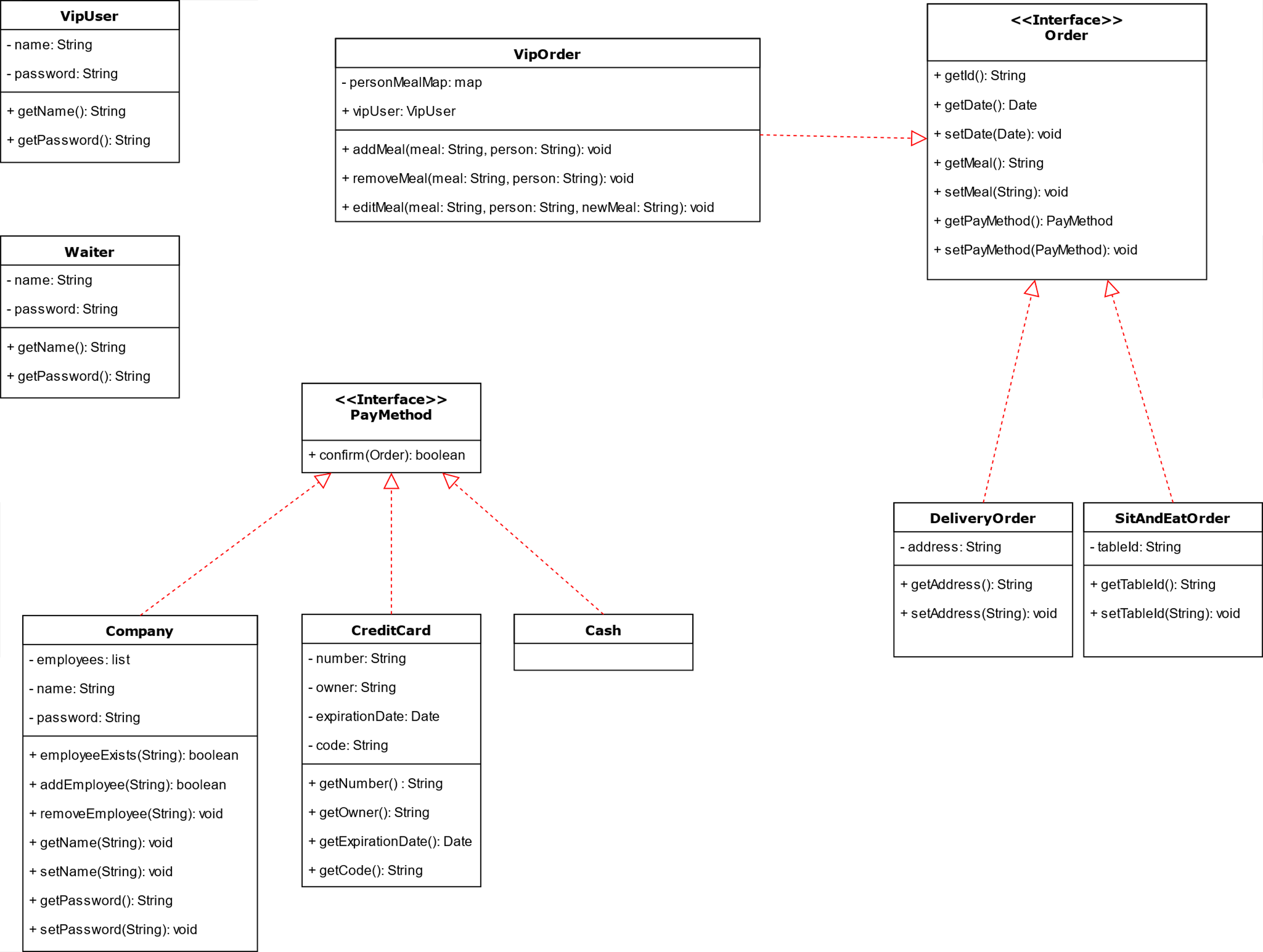


כלומר, העובד יכיל Interface לכתיבת מידע. כאשר נרצה לכתוב מידע נקרא לפונקציה write של Interface זה. בעת יצירת העובד נממש interface זה בעזרת אחד מבניו. במידה ונכניס יכולת חדשה רק נצרך ליצור מחלקה חדשה שיורשת ממנו וזהו.

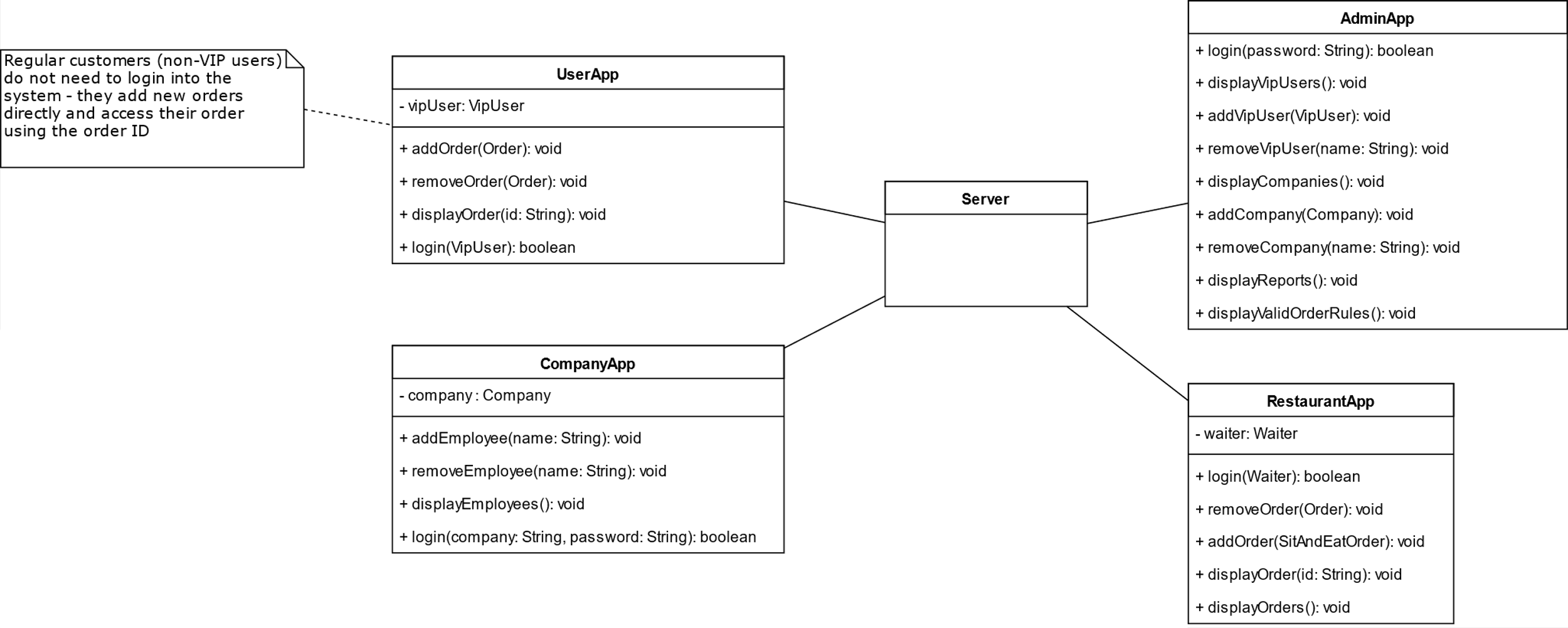
1. העקרון המופר הוא עקרון ההחלפה של ליסקוב. למרות שנשמע הגיוני שריבוע יירש ממלבן מכיוון שריבוע הוא סוג של מלבן, בפועל למלבן יהיו מתודות אשר אין בהן צורך בריבוע, למשל setWidth ו-setWidth, כאשר ריבוע צריך רק אחת מהן.

בפועל קיים מצב בו אם נקבל ריבוע במקום מלבן לא נוכל להשתמש בו כהלכה.

1. אחד היתרונות של הdesign pattern – Factory הוא האפשרות להוסיף אובייקטים חדשים למערכת מבלי שנדרש לשנות את תכן המערכת, לכן מקיים בדיוק את עקרון הפתחיות/סגירות. בנוסף עקרון היוצר מ-GRASP מגדיר איפה ומתי נרצה לממש יצירת אובייקט. נשים לב כי פונקציית הfactory מקיימת חלק מתנאים אלו, למשל, יש לה את המידע הנדרש על מנת לאתחל מופעים.
2. הממשק PayMethod הוא Information Expert במובן שהוא אחראי על אישור טרנזאקציות כספיות של הזמנות מהמסעדה. המחלקה Order של הזמנה מכילה שדה של PayMethod אך אין לה מספיק מידע בשביל לאשר טרנזאקציה:



המחלקה AdminApp היא יוצרת (ומוחקת) מופעים של VipUser:



1. הdesign pattern שמומש כאן הוא factory כי יצירת האובייקט מתרחשת בתת מחלקה ולפי דרישה ולא באופן ישיר ומפורש.
2. נראה שהdesign pattern שמומש הוא singleton שכן נראה שלא ניתן לייצר יותר מSecurityManager אחד וכל ניסיון ליצור אחד נוסף פשוט יחזיר את הinstance הקיים.

שאלה 3

הdesign patterns השונים מומשו באופן הבא:

Singleton – המחלקה ColorGenrator היא סינגלטון. המימוש הינו בדומה להרצאות על ידי הפיכת פונקציית הבנאי לפרטית ויצירת פונקצייה סטאטית שדואגת לקיום של מופע יחיד של המחלקה.

בזכות שימוש בסינגלטון אנחנו יוכלים להיות בטוחים שכל Billboard במערכת ישנה את צבעו לאותו צבע.

Observer – הObserver במקרה זה היא המחלקה Billboard והObservable היא המחלקה ColorGenerator.

יש צורך בdesign pattern זה מכיוון שכל הלוחות צריכים לדעת שהגיע הזמן לשנות את צבעם ולכן עליהם להקשיב לColorGenerator.

המימוש של design pattern זה הוא על ידי המחלקות המובנות של JAVA.

Billboards יורש מObserver ו-ColorGenerator יורש מObservable.

ניתן להוסיף כמה לוחות שנרצה בעזרת קריאה לפונקציה addObserver של הObservable, וכל שתי שניות יעבור הColorGenerator ויודיע לכולם שהגיע הזמן להחליף צבע ומהו הצבע החדש בעזרת הפונקציה notifyObservers. לאחר התראה זו יעבור כל Billboard על מערך הPanels שלו ויעדכן אותם לפי הStrategy שנבחרה.

Strategy – השתמשנו בdesign pattern זה על מנת לאפשר חופש בסדר עדכון הPanels בכל Billboard מבלי ליצור את כל המחלקה מחדש.

Design pattern זה מומש על ידי יצירת interface בשם ColorChangeStrategy עם פונקציה בודדת GetColorChangeOrder. כאשר ניצור Billboard נצתרך לספק מחלקה כלשהי אשר מממשת interface זה.

כך נוכל תחת אותו ממשק של Billboard לממש סדרי עדכון שונים בקלות.