עקרון dependency inversion – בהינתן מחלקה B, שממשמת אינטרפייס I, המחלקה A, תחזיק אובייקט מסוג האינטרפייס, זה בדיוק ההיפוך הinversion, כי ניתן להחליף את האובייקט מסוג האינטרפייס, זה לא בהכרח יהיה B. השליטה עברה למי שאנחנו תלוי בו כלומר בB. אנחנו רואים פה strategy pattern, הsource קוד לא משתנה אבל התלות בזמן ריצה משתנה בI

Dependency injection – הזרקה של תלות. כאשר נרצה להפעיל איזה שהיא מתודה שנמצאת בA, ואנחנו בעצם במתודה מקבלים מישהו שהוא מטיפוס האינטרפייס, אנחנו ממש בקוד צריכים לומר איזה אובייקט שיורש או ממש את האינטרפייס יהיה במתודה הזאת שנפעיל. זו ממש הזרקה של התלות שלנו.

Inversion of control – IOC – היפוך של השליטה, השליטה הופכת להיות אצל מי שמפעיל את הקוד. בהינתן יצירת אובייקט אנחנו רוצים להפעיל את המתודות הפומביות של אותו אובייקט וזה נמצא בעצם ביכולת של המפעיל להחליט מתי הוא מפעיל אותם. זה לא נמצא אצל מי שכותב את הקוד.  
בילדר פטרן – הפרדה בין החלקים של היצירה לבין מי שמחליק איך להרכיב אותם. סטרינג בילדר ממש את הבילדר פטרן, הוא מייצר מחרוזת שהיא דינאמית וניתן להוסיף לה דברים לסוף. ההמתודה append מחזירה את אותו string builder. מתי נהפוך את זה למחזורת בנייה, .TOString() הוא החזיר כבר משהו שלא ניתן לשנותו.  
השליטה נמצאת אצל מי שמפעיל את הקוד. השליטה נמצאת אצל מי שמשתמש בקוד ולא מי שהחליט לכתוב את הפטרן של בילדר או את toString().

למה נאהב אובייקטים שלא ניתן לשנותם – immutable, זה ממש טוב כמפתחות, הם לא מייצרים לנו בעיות. Conceracy – מה הקשר לאובייקטים שהם אימיוטבל – אין סכנה שיהיו כמה תרדים שיגעו וישנו אותו. שימוש בfinal מושם פעם אחת בלבד. כאשר כל השדות הם פיינל אני מחוייב לאתחל אותם בבנאי של המחלקה.

נרצה לאפשר לקליינט איזה שדות לבחור למרות שכולם final, איך הבילדר פטרן יכול לסייע לנו. אנחנו לא רוצים ליצור מלא בנאים עם כל האופציות האפשריות זה 2 בחזקת n המשתנים שעלינו לאתחל. נצטרך מחלקה שיקראו לה יוזר בילדר, השדות שיש לה זה אותו דבר כמו יוזר רק שהשדות אינם פיינל. כל מתודה שמזינה ליוזר בילדר איזה שהוא שדה היא תחזיר יוזר בילדר – כי אז ההפעלה הבאה תעבוד על אותו יוזר בילדר, ובסוף יש את המתודה בילד שתחזיר יוזר עם כל השדות הפיינל.

הקונסטרקטור של יוזר מקבל יוזר בילדר, וכל השדות של יוזר שווים לשדות של אותו יוזר בלידר. בהנחה שיש לנו כזה כמובן. נשים לב בתוך מחלקת יוזר נכתוב תת מחלקה שיקראו לה יוזר בילדר. למה המחלקה הזאת תהיה פומבית וסטטית, כדי שנוכל לקרוא לה מבחוץ, כן דרך המחלקה אבל לא דרך אובייקט. היא סטטית כי היא שייכת למחלקה יוזר אבל לא לאובייקט. מה עוד נאמר על השדות של יוזר בילדר, מי יהיה פיינל רק השם והשם משפחה כי הם הrequiered. כעת יש לנו מתודות נוספות ביוזר בילדר, שכל מתודה מאתחלת את השדה הרלוונטי וכל מתודה כזאת מחזירה לנו יוזר בילדר כדי שנוכל לשרשר את הפעולות. ממש בסוף נקרא למתודה הטרמינלית שהיא תחזיר את היוזר. למה מותר להפעיל את הקונסטרקטור של יוזר, כי אנחנו בתוך המחלקה ויכולים להפעיל את כל מה שקשור למחלקה.

איך נקבל יוזר חדש? מותר ליצור יוזר בילדר חדש, הוא פומבי וסטטי, הבנאי של יוזר בילדר חייב לקבל שם ושם משפחה אז כעת ביד יש לנו יוזר בילדר. בתוך יוזר בילדר נפעיל את המתודות השונות שמחזירות יוזר בילדר כדי שנוכל לעבוד עם אותו יוזר בילדר בבנאי של המחלקה יוזר האימיוטבילית שאותו בעצם רצינו ליצור.

הבילדר פטרן הוא מעין גשר, יש כל מיני חלקים לבנייה, ובסוף את המתודה build, כל אחד מימש את החלקים בדרכו שלו, ואז יש כל מיני אובייקטים קונקטרים שמימשו את הבילדר פטרן. כעת יש כל מיני אובייקטים שמימשו את אלגוריתם היצירה באופן שהם רצו עם אותם תכנים שהם רצו להכניס בבנייה.

Mvc – יש לנו את הלוגיקה העסקית שקורית במודל ויש את הלוגיקה של הפרזנטציה שקוראת בתצוגה, אבל הלוגיקה של ההפעלה, המדיוניות כיצד להפעיל את המודל והתצוגה תימצא בקונטרולר. כדי ליצור שולט כזה צריך לממש את העקרונות של אינברשין אוף קונטרול. בניגוד לקונטרולר ויו מודל הוא לא קונטרולר, עבור המודל הוא הויו, עבור הויו זה המודל שלו. הויו יכול לדבר עם הויו מודל בלי לדעת איך זה יעבוד אחכ. מה מייחד את MVVM – הdata binding.  
איך נייצר data binding שכזה – עם האובזרבר פטרן.

צרו את המחלקה פרופרטי שפועלת על פי המחלקה V. כל פרופטי יכול לעשות בינד לאחר כאשר אחד משתנה גם חברו משתנה. פרופטרי המחלקה צריכה להיות גם observable – שכל פעם שיעשו לה set, מצד שני היא צריכה גם לצפות כאשר עושים שינוי. כאשר עושים לה set היא צריכה להודיע לכל מי שצופה בה. היא תעשה notify לכולם, מה צריכה לעשות update, בכל אחד מהצופים, הם צריכים להוסיף את עצמם כobsever. מכיוון שהם הוסיפו את עצמם כצופים כאשר יעשו להם notify , הם ידעו על השינוי.

ההבדל בין mvvm and mvc - נדמיין ויו עם המון המון מרכיבים, אם היה לנו mvc עם קונטרולר, שישאל את הויו מה קרה ואז הקונטרולר צריך לומר מה לעשות בתגובה. לעומת זאת עם היו לנו פרופרטיז לכל אותם אופציות שיכול להיות בהם שינוי, נוכל להפעיל פשוט בינדינג שאם קורה משהו בויו זה ישתנה בויו מודל ובמקביל ישתנה גם במודל. כזאת יעשה בעזרה אירועים = איבנטים, שכאשר נעשה שינוי אז נפעיל פונקציה מסוימת שאנחנו כן יודעים **מה**  צריך לקרות אבל אנחנו לא יודעים **מתי** זה צריך לקרות.

הויו כורך את תכונותיו לתכונות של הויו מודל.

String pool – בכל השפות שיש להם סביבה שמנהלת את הקוד שלנו סביר שיש להם דבר כזה. כל פעם שנייצר מחרוזת, זה נשמר בpool שלנו. איך נדע אבל באיזה כתובת שהיא נמצאת? לכל סטרינג יש את המתודה intern(), שמוביל אותנו לכתובת של אותה מחרוזת.  
נגיד שייצר  
string s1 = new string(other existing string)

כעת s1 מחזיקה את אותה מחרוזת והאינטרן שלה מצביע לאותה כתובת של אותה מחרוזת שגם האינטרן שלה מצביע לשם.  
אם נעשה s1.equals(s2) – זה יעלה לנו את אורך המחרוזת וזה יחזיר true. אם נעבוד עם האש אנחנו נוכל לבצע את ההשואה בo(1).

אם האינטרן לא מאותחל, כלומר null, נצטרך להכניס אותו לpool. מה לגבי המחזורת שהיא העתק של המחזורת הראשונה, ההשוואה הראשונה עלתה או של 1, אבל כל החיפוש וההכנסה של הפול, עדיין עלו או של אנ. כלומר שווה להשתמש בpool, זה מאפשר השוואת כתובות בo(1). נזכור את העניין שסטרינג היא אימיוטבל.

**המבחן!!**

שעתיים וחצי, 4 שאלות. 2 שאלות של תכנות ו2 של בקיאות בחומר. עונים בגוף השאלון. חומר סגור.

נעדיף שהפלט של עורך ויזואלי יהיה XML, לא נרצה שהעורך הויזאלי יג'נרט לנו קוד.

לכל סטרים יש תרד משלו, לכל סטרים יש באפר. בתדר שלו הוא עובר על כל האיברים בבאפר ועושה להם משהו – במקרה של השאלה במבחן הוא עושה עליהם את הפונקציה של המאפ. את כל האיברים האלו אנחנו צריכים להעביר לבאפר של הסטרים הבא.

כל יצירת הסטרים צריכה להיות בתרד משלה, כאשר כל הקלט יהיה מוכן והסטרים יהיה מוכן זה יגיע לתחנה הבאה. צריך להיות תרד שמריץ לולאה.