

חלק 13

תכנות מונחה עצמים -

נושאים נוספים וסיכום

נושאי הפרק

- למה תכנות מונחה עצמים?
- עמידות לשינויים
- מודולריות
- שימוש חוזר בתוכנה
- למה תיכון בעזרת חוזים?
- עלות תיקוני שגיאות (שמתגלות באיחור).
- עדיף לכתוב קוד נכון מלכתחילה
- תבניות תיכון - לתיאור מבנים נפוצים, לפתור בעיות שחוזרות
- refactoring - לשיפור מבנה התכנית
- תיכון בעזרת חוזים - סיכום וחזרה.

עמידות לשינויים

- מערכות תוכנה מתקיימות שנים רבות, במיוחד מערכות משובצות (לדוגמא מטוסים).
- עלות התחזוקה (תיקוני שגיאות מאוחרים, הוספת תכונות, התאמה לשינויי טכנולוגיה) היא יותר ממחצית העלות הכוללת של מחזור החיים.
- לכן השקעה נוספת בשלבים המוקדמים שתקטין את עלות התחזוקה עשויה להיות כדאית
- לכן חשוב שהתוכנה תהיה קריאה, ומודולרית.
- כמו כן התיכון צריך לאפשר שינויים עתידיים צפויים.
- אבל קשה כמובן לצפות.

מודולריות

- מודולריות היא תכונה חשובה של תוכנה.
- נחוצה כדי לאפשר הפרדת עניינים בזמן הפיתוח, ולשפר קריאות לצורך תחזוקה.
- מודולריות פירושה היכולת לפרק מערכת למרכיבים, לבנות מערכת ממרכיבים, להבין כל מודול בפני עצמו, רציפות, הגנה
- מודולריות טובה כתכונה של מערכת דורשת מודולים בעלי חוזק פנימי גבוה, וצמידות נמוכה.
- ארכיטקטורת מערכת שמבוססת על הנתונים מאפשרת מודולריות טובה יותר מארכיטקטורה שמבוססת על הפונקציונליות.
- מכאן היתרון של פיתוח תוכנה מונחה עצמים.

שימוש חוזר בתוכנה

- על מנת לשמור על עלויות תוכנה סבירות, יש לשפר את תפוקת מפתחי התוכנה.
- שיפור תפוקה יומית של מתכנת דורש שיפורים משמעותיים בתהליכי הפיתוח, שפות התכנות, וכלי הפיתוח.
- בנוסף, ניתן להקטין את עלות הפיתוח ע"י שימוש ברכיבי תוכנה קיימים, שפותחו עבור פרויקט קודם או פותחו במיוחד כתשתית לארגון.
- שימוש חוזר בתוכנה כרוך בקשיים רבים, לא כולם טכניים: תסמונת "לא הומצא אצלנו", תשלום עבור תוכנה לפי שורת קוד
- הניסיון מראה שרכיבי תוכנה מונחת עצמים מתאימים לשימוש חוזר יותר מרכיבים פרוצדורליים.

תבניות תיכון - מוטיבציה

בחיי יום יום אנחנו מתארים דברים תוך שימוש בתבניות חוזרות:

● מכונית א' היא כמו מכונית ב', אבל יש לה 2 דלתות במקום 4.

● אני רוצה ארון כמו זה, אבל עם דלתות במקום מגרות.

גם בפיתוח תוכנה, אנחנו יכולים להסביר כיצד לעשות משהו ע"י התייחסות לדברים שעשינו בעבר, ובצורה כזאת להקל על התקשורת עם עמיתים.

● נאחסן את המבנה בעץ בינרי, ונבצע חיפוש לרוחב.

הגדרות מהספרות:

- "Design Patterns are recurring solutions to design problems you see over and over." [Alpert, Brown, Wof, 1998].
- "Design Patterns constitute a set of rules describing how to accomplish certain tasks in the realm of software development." [Pree, 1994].
- "A pattern address a recurring design problem that arises in specific design situations and presents a solution to it." [Buschmann et al, 1996].
- "Patterns identify and specify abstractions that are above the level of single classes and instances, or of components." [Gamma et al, 1993].

מהי תבנית תיכון?

- תבנית תיכון היא פתרון מקובל לבעית תיכון נפוצה בתכנות מונחה עצמים.
- תבנית תיכון מתארת כיצד לבנות מחלקות כדי לענות על הדרישה הנתונה.
- מספקת מבנה כללי שיש להשתמש בו כשממשים חלק מתכנית.
- לא מתארת את המבנה של כל המערכת.
- לא מתארת אלגוריתמים ספציפיים.
- מתמקדת בקשר בין מחלקות.
- מתארת נסיון מצטבר של מתכננים, שניתן ללמוד ועוזר לתקשורת בין מהנדסי תוכנה.

מקורות

- המושג נעשה פופולרי בעקבות הספר שמכונה GoF :

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Design Patterns: Elements Of Reusable Object-Oriented Software. 1995.

- הגישה אומצה מעבודותיו של כריסטופר אלכסנדר, מתחום ארכיטקטורה של מבנים. הוא כתב :

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

רקע

- אבחנותיו של אלכסנדר רלבנטיות גם למערכות תוכנה.
- במקום לדבר על קירות ודלתות, אנו עוסקים בעצמים ומנשקים.
- נעשה מאמץ לקטלג תבניות תיכון כלליות, וגם תבניות שמתאימות לתחומי יישום מוגדרים.
- הקריטריון לקבלת תבנית תיכון - נעשה בה שימוש במספר יישומים אמיתיים.
- סוגים נוספים של תבניות, לתיעוד דרכים טובות לפתור בעיות מסוגים שונים (כולל למשל ניהול כוח אדם). Best Practice
- אנטי תבניות (Anti patterns) דברים שיש להימנע מהם.

מרכיבים עיקריים של תבנית תיכון

ארבעה מרכיבים חיוניים:

- שם התבנית. שימוש בשם אחיד ומקובל מסייע לתקשורת בין מהנדסי תוכנה, מעלה את רמת ההפשטה.
- הבעייה. מתי ניתן להשתמש בתבנית? איזה בעייה היא אמורה לפתור? יכול לכלול מבנה עצמים שהוא סימפטום לבעייה, או אוסף תנאים שצריכים להתקיים.
- הפתרון. מתאר את מרכיבי הפתרון, היחסים ביניהם, אחריות כל אחד ושיתופי פעולה. תבנית לפתרון, ולא מימוש קונקרטי.
- השלכות. תוצאות וקשרי גומלין (trade-offs) של שימוש בתבנית. חשוב לצורך הערכה של חלופות תיכון והבנת העלות והתועלת של התבנית. השלכות על יעילות, גמישות, יכולת הרחבה ויבילות (portability).

מושגים שקשורים לתבניות תיכון

תבניות תיכון מטפלות ברמת ביניים בין שני סוגי התבניות הבאים (עפ"י Buschmann and al):

- תבניות ארכיטקטוניות: צורת ארגון בסיסית של מערכת תוכנה שלמה. מספקת אוסף תתי מערכות קבועות מראש, חלוקת אחריות, כללים והנחיות לארגון היחסים ביניהן.

- תבניות קידוד, או idioms. תבנית ברמה נמוכה, ספציפית לשפת תכנות מסוימת. מתאר איך לממש היבט מסוים של רכיבים ויחסים ביניהם בעזרת המבנים ששפת התכנות מספקת.

תבניות תיכון אמורות להיות בלתי תלויות בשפה, אבל התבניות של GoF (וגם אחרות) מתייחסות במיוחד לשפות מונחות עצמים.

תבניות תיכון ו Frameworks

Design patterns focus more on reuse of recurring architectural design themes, while frameworks focus on detailed design ... and implementation." [Coplien, Schmidt, 1995]

- Framework (OO Software) הוא אוסף מחלקות קשורות זו לזו, שניתן להרחיב ו/או ליצור מופעים, כדי לממש יישום.
- זה מגדיר ארכיטקטורת תוכנה ניתנת לשימוש חוזר, שמספקת מבנה והתנהגות כלליים של משפחה של יישומים ממוחשבים.

תבניות תיכון ו Frameworks - המשך.

- מספק מידע נוסף על שיתופי הפעולה בין המחלקות והשימוש בהן בתחום יישום מסוים.
- זה אינו יישום מלא שניתן להריץ, כי חסרים חלקים שיש להוסיף או להחליף כדי לקבל את הפונקציונליות הדרושה.
- תבניות תיכון יכולות לתאר את הקשרים בין מחלקות ב Framework
- Framework יכול להשתמש במספר תבניות תיכון

קלסיפיקציה של תבניות תיכון

הספר של GoF מציג 23 תבניות שמחולקות לשלוש משפחות לפי המטרה שלהן:

- תבניות ליצירה Creational: נוגעות לתהליך היצירה של עצמים.

- תבניות מבנה Structural: עוסקות בהרכבה של מחלקות ועצמים.

- תבניות התנהגות Behavioral: מאפיינות את הדרכים בהן מחלקות ועצמים מתקשרים ומחלקים אחריות.

קלסיפיקציה נוספת מתייחסת לתחום העיסוק של תבנית - מחלקות או עצמים.

בנוסף, ניתן להתייחס לקבוצות של תבניות שמופיעים בדרך כלל ביחד, או כאלה שמהווים חלופות שונות לפתרון בעיות דומות.

Model View Controller

- גישה לבניית מנשקים גראפיים שהוצעה בסמולטוק 80, ואומצה גם ע"י מפתחים בשפות וסביבות אחרות.
- יישום בנוי משלושה סוגים של עצמים:
 - מודל Model: עצם של היישום.
 - מראה View: הצגה של המודל על המסך.
 - בקר Controller: מגדיר את הדרך שבה המנשק מגיב לקלט של המשתמש.
- גישה זאת עדיפה על טיפוך בשלושת המרכיבים האלה ביחד.
- MVC משפר את הגמישות ואת השימוש החוזר.

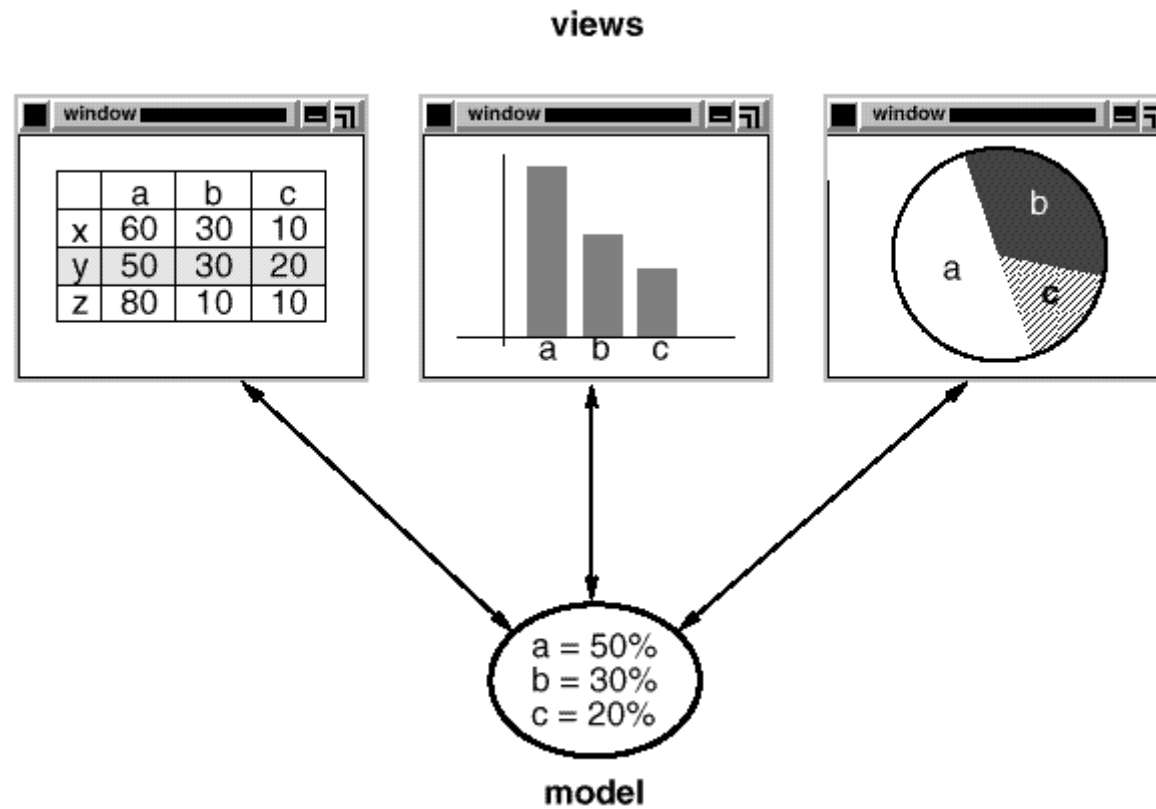
מודל לעומת מראה

- נניח בינתיים שאין בקר.
- למודל יכולים להיות מספר מראות.
- רוצים ליצור הפרדה בין המראה למודל באמצעות פרוטוקול:
- המראה שולח ראשית בקשה להתחבר למודל (להירשם כמנוי).
- כאשר המצב של המודל משתנה, הוא שולח לכל המנויים הודעה על השינוי.
- כל מראה מעדכן את עצמו.
- נשתמש בתבנית Observer (יש לה גם שימושים נוספים)

תבנית התיכון Observer

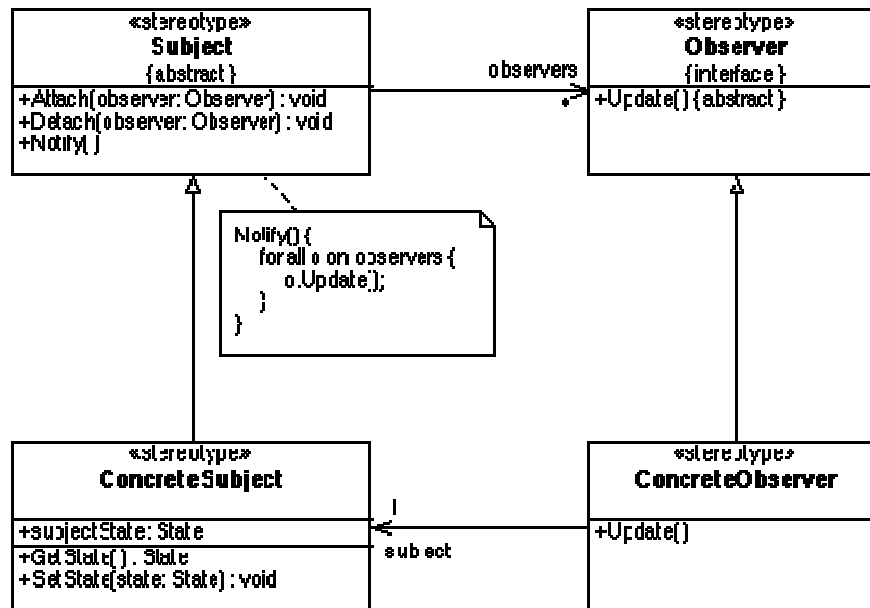
- מטרה: להגדיר תלות של אחד לרבים בין עצמים, כך שכאשר האחד משנה את מצבו, העצמים התלויים מקבלים הודעה ומתעדכנים אוטומטית. (תבנית התנהגות).
- מוטיבציה: לשמור על עקביות בין עצמים קשורים, בלי לגרום לצימוד חזק מדי.
- ישימות: כאשר
- להפשטה יש שני הבטים, אחד תלוי בשניץ עצמים נפרדים מקלים על שינוי ושימוש חוזר.
- שינוי בעצם אחד דורש שינוי במספר לא ידוע של עצמים.
- עצם יכול להודיע לעצמים אחרים בלי להניח משהו עליהם.

תבנית התיכון Observer - מוטיבציה



תבנית התיכון Observer - מבנה

נשתמש בדיאגרמת מחלקות

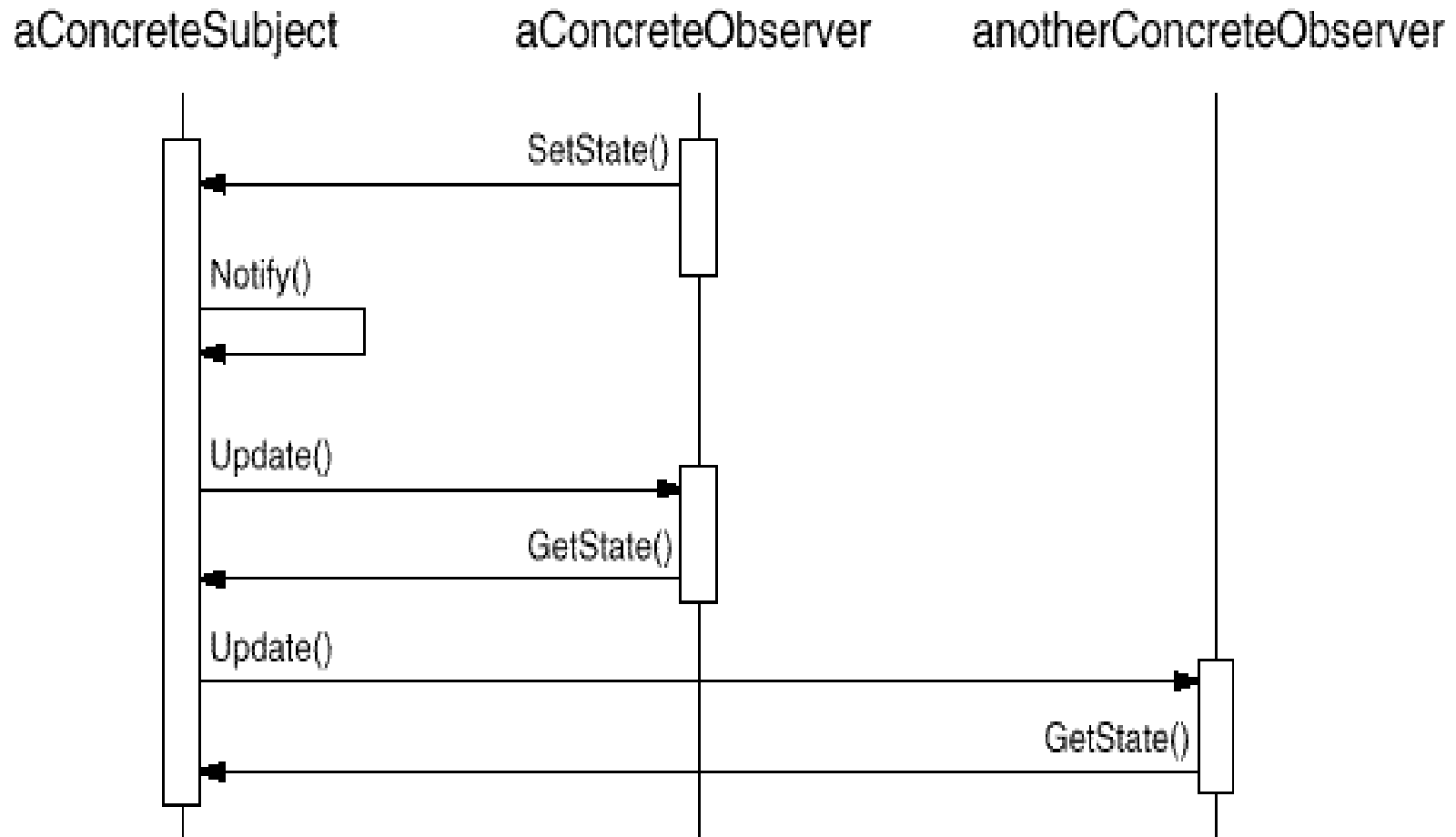


תבנית התיכון Observer - משתתפים

- Subject - שומר רשימה של Observers. מייצא פעולות להוספה והשמטה של Observers. כולל פעולת notify ששולחת הודעת עדכון לכל ה Observers.
 - Observer - מייצא פעולה מופשטת של עדכון.
 - ConcreteSubject - יורש מ Subject, שומר מצב, קורא ל notify כאשר המצב משתנה.
 - ConcreteObserver - יורש מ Observer, מחזיק התייחסות ל ConcreteSubject, מממש את פעולת העדכון.
- הערה (לגבי כל תבניות התיכון): שמות המשתתפים בתבנית מתארים את תפקידי המחלקות בתבנית. שמות המחלקות במערכת יהיו בדרך כלל שונים, ויבטאו את התפקיד הכללי יותר של המחלקה.

תבנית התיכון Observer - שיתוף פעולה

נשתמש ב sequence diagram



Model View Controller - המשך

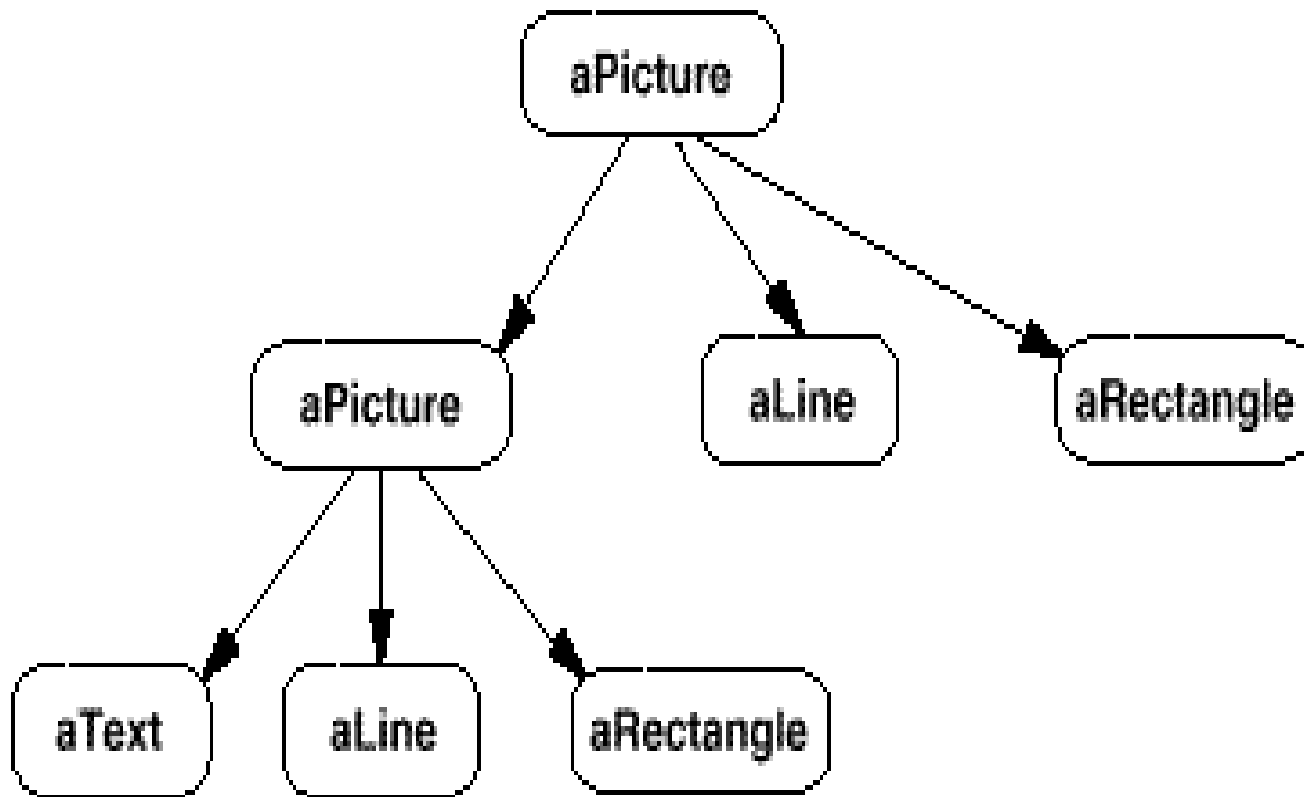
- מראות (Views) יכולים להיות מקוננים.
- לדוגמא, לוח בקרה מכיל כפתורים.
- מראה מורכב מתנהג בדיוק כמו עצם מראה רגיל.
- נשתמש בתבנית התיכון Composite (שימושית בהרבה הקשרים נוספים).

תבנית התיכון Composite

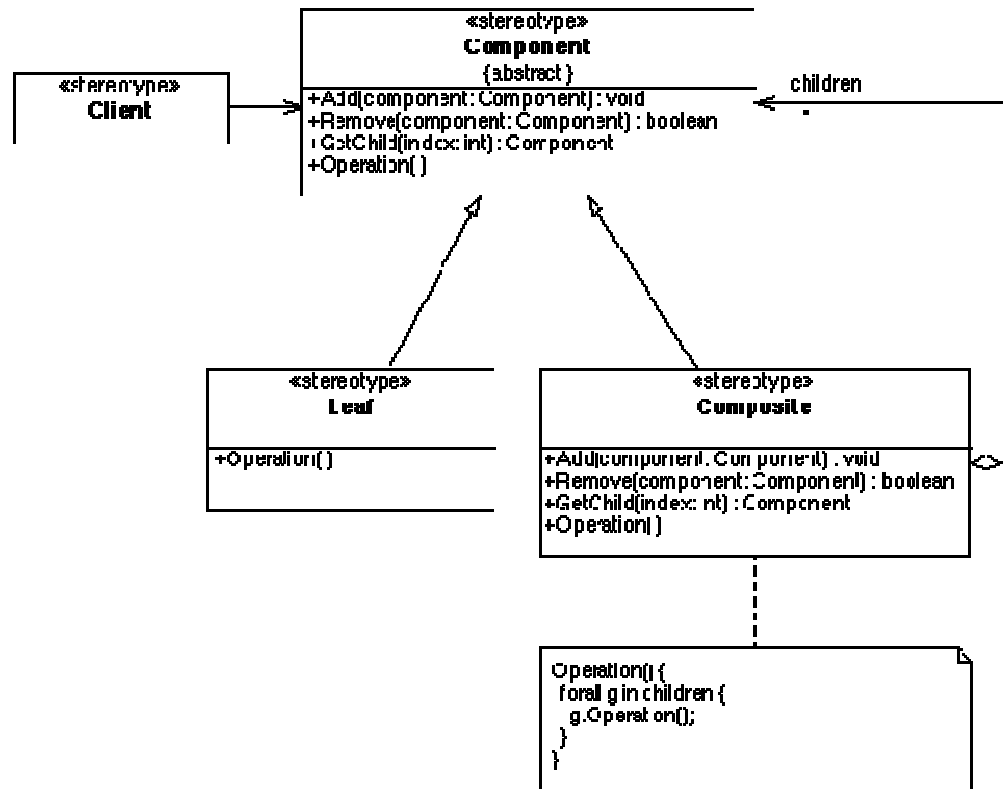
- מטרה: להרכיב עצמים למבני עץ שמייצגים את ההיררכיה של היחס חלק-שלם. לאפשר ללקוחות לטפל בעצמים בודדים ובהרכבות באופן אחיד. (תבנית מבנה).
- דוגמא: עצמים גרפיים.
- ישימות:
- כאשר רוצים לממש יחס חלק-שלם.
- כאשר רוצים לאפשר ללקוחות להתעלם מהבדלים בין עצמים מורכבים לעצמים בודדים.

תבנית התיכון Composite - דוגמא

דוגמא למבנה של ציור



תבנית התיכון Composite - מבנה



תבנית התיכון Composite - משתתפים

- Component - מגדיר מנשק לעצמים בהרכבה. מיישם ברירת מחדל לפעולות המשותפות. מגדיר מנשק לגישה לרכיבים הילדים.
- Leaf - (אחדים) - יורש מ Component. מייצג רכיבי עלה. מיישם התנהגות לעצמים הפרימיטיביים.
- Composite (אחדים?) - יורש מ Component. מייצג רכיבים מורכבים. מיישם פעולות על הילדים ע"י איטרציה על רשימת הרכיבים הכלולה בו.
- Client - לקוח של Component, מבצע פעולות על עצמים רק דרך מנשק ה Component.

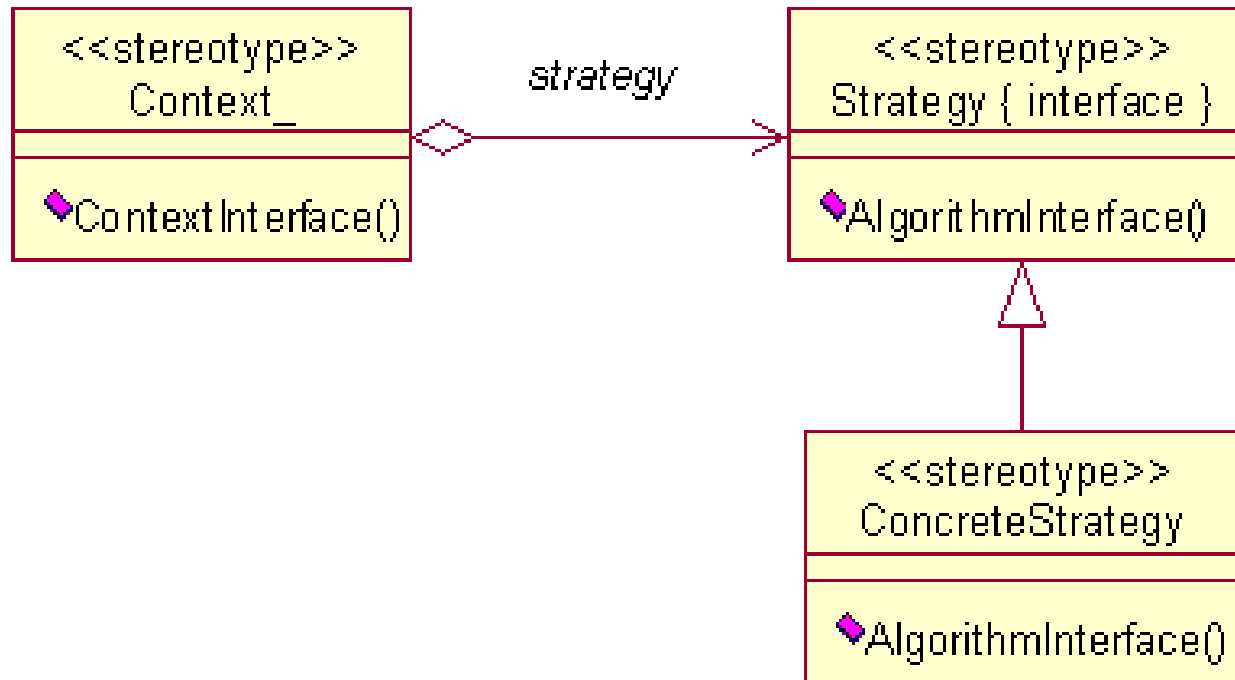
Model View Controller - המשך

- הבקר משמש להכמסה של מנגנון התגובה: איך מראה מגיב לקלט של המשתמש.
- מאפשר שנויים בצורת התגובה בלי לשנות את המראה.
- לדוגמא, החלפת כפתורים לפקודות בתפריט.
- שינוי כזה אפשרי אפילו בזמן ריצה.
- היחס בין מראה לבקר הוא דוגמא לתבנית התיכון Strategy .

תבנית התיכון Strategy

- מטרה: לספק הכמסה למשפחה של אלגוריתמים ולעשותם ברי החלפה. לאפשר לאלגוריתמים להשתנות באופן בלתי תלוי בלקוחות. (תבנית התנהגות).
- מוטיבציה:
- הסרת האלגוריתם מפשטת את הלקוחות.
- מדיניות שונה בזמנים שונים.
- ישימות:
- כאשר אוסף מחלקות שונות זו מזו רק בהתנהגות.
- יש צורך בחלופות (למשל שוני בזמן/זכרון).
- האלגוריתם צריך להסתיר נתונים מורכבים.
- להוציא התנהגות מותנית מהמחלקה.

תבנית התיכון Strategy - מבנה



תבנית התיכון Strategy - משתתפים

- Strategy - מגדיר מנשק משותף לכל האלגוריתמים הנתמכים.
- ConcreteStrategy (אחדים) - מספק מימוש של האלגוריתם בהתאם למנשק Strategy
- Context - לקוח של Strategy . בזמן ריצה, ההתייחסות היא לעצם מטיפוס של אחד מ ConcreteStrategy .

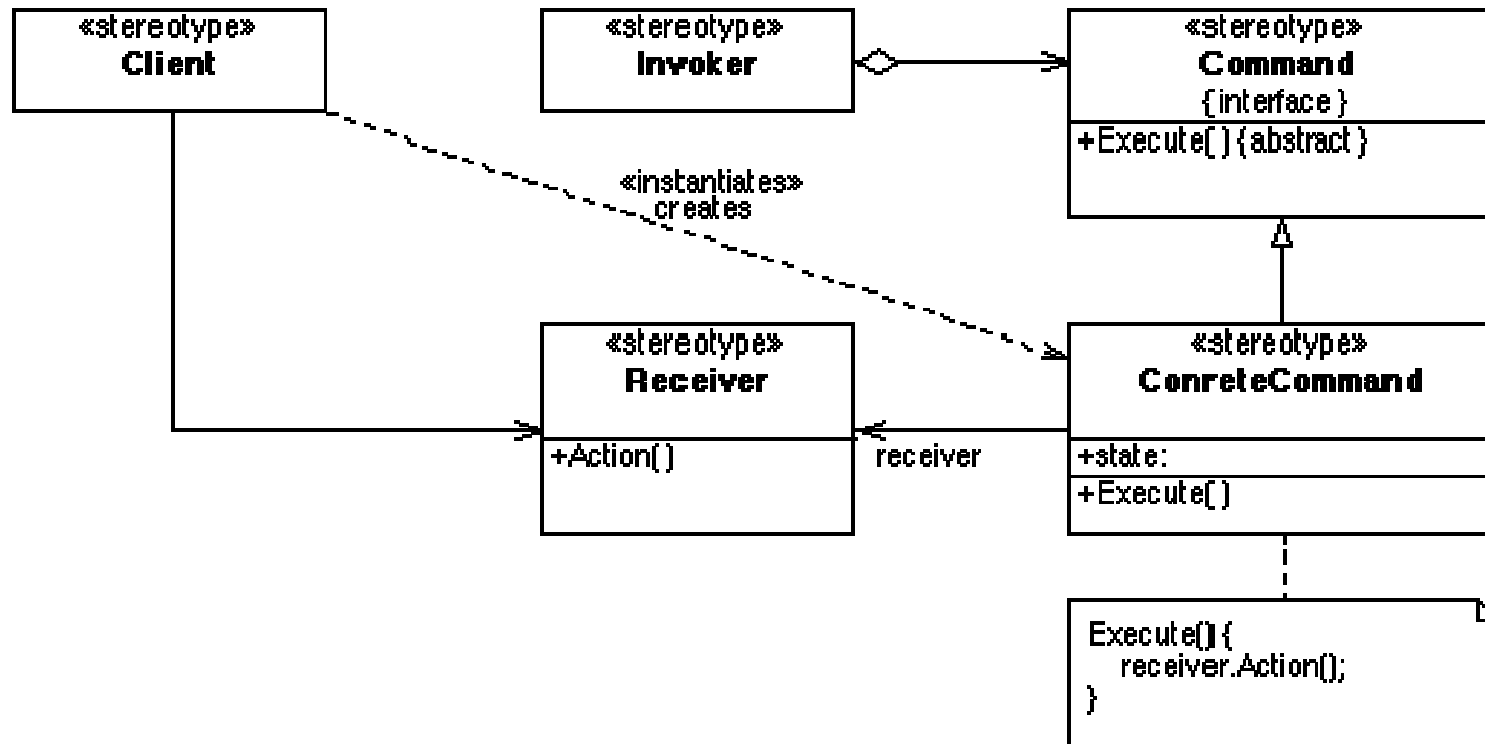
תבניות תיכון ליצירה

- תבנית התיכון Singleton נועדה להבטיח שיש רק אובייקט אחד ממחלקה מסוימת. ראינו איך ליישם זאת בג'אווה.
- דברנו גם על בית חרושת, גם זאת תבנית תיכון ליצירה.
- למעשה יש כמה תבניות תיכון העוסקות בבית חרושת. (למשל תבנית התיכון Abstract Factory).

תבנית התיכון Command

- מטרה: להגדיר מנשק לצורך ביצוע פעולה. (תבנית התנהגות).
- ישימות:
- פרמטריזציה של עצמים על פי הפעולה שיש לבצע. תחליף מונחה עצמים למנגנון של callback .
- לאפיין, לצבור ולבצע בקשות בזמנים שונים, אולי בתהליך נפרד.
- תמיכה ב undo ו redo
- לשמור רשימת פעולות לביצוע כאשר מחלימים מקריסת מערכת.
- לבנות מערכת סביב פעולות ברמה גבוהה הבנויות מפעולות פרימיטיביות (טרנסקציות).

תבנית התיכון Command - מבנה



תבנית התיכון Command - משתתפים

- Command - מגדיר ממשק לביצוע פעולה.
- ConcreteCommand - יוצר קשירה בין עצם Receiver לבין פעולה. ממש שרות execute ע"י הפעלת פעולה או פעולות של ה Receiver.
- Client - יוצר עצם ConcreteCommand וקובע את ה Receiver שלו.
- Invoker - מבקש מה Command לבצע את הפקודה המתבקשת.
- Receiver - יודע כיצד לבצע את הפעולות הדרושות לביצוע פקודה מבוקשת. כל מחלקה יכולה להיות Receiver.

סיכום תבניות תיכון

- פתרון מקובל לבעית תיכון נפוצה בתכנות מונחה עצמים.
- מבנה כללי שיש להשתמש בו כשממשים חלק מתכנית.
- נסיון מצטבר שניתן ללמוד ועוזר לתקשורת בין מהנדסי תוכנה.
- ראינו חלק קטן מהתבניות מהספר של GoF .
- יישום לדוגמא בשפת תכנות נתונה (למשל ג'אווה).
- בעשור האחרון הוצעו כמה מאות תבניות, לא כולן חשובות באותה מידה.
- השימוש בתבניות חדר גם למושגים אחרים בהנדסת תוכנה, וגם בתחומים אחרים.
- תבניות ואנטי תבניות.

refactoring

- refactoring הוא תהליך של שינוי תוכנה כך שהתנהגותה החיצונית לא תשתנה, אך המבנה הפנימי שלה ישתפר.
- לנקות ולשפר את הקוד בלי להכניס לשגיאות.
- "שיפור התיכון אחרי שהקוד נכתב" סותר לכאורה את העקרונות שמנחים פיתוח תוכנה.
- אבל מכיר בעובדה שבמשך הזמן, שינויים בקוד (למשל להוספת תכונות) גורמים לכך שהמבנה נפגע ומסתבך.
- ב refactoring מבצעים בכל פעם שינוי קטן, טרנספורמציה שמשמרת נכונות (כלומר לא משנה את ההתנהגות החיצונית).
- לאחר כל שינוי יש לבדוק היטב שהשינוי היה נכון - להריץ את אוסף הבדיקות שצברנו.

מקורות

- האנשים שזיהו את חשיבות הרעיון :

Ward Cunningham, Kent Beck

- ספר:

Martin Fowler, Refactoring, Improving the Design of Existing Code, Addison Wesley 2000. (2nd edition 2005).

- קשור ל Extreme Programming - תהליך פיתוח חדש יחסית, ששם דגש על פיתוח אבולוציוני, גירסאות תכופות, בדיקות, קשר הדוק עם הלקוח, קידוד בזוגות, ...

- משפחת תהליכי פיתוח Agile Software Development

למה refactoring ?

- לשפר את תיכון התוכנה - אחרת מבנה המערכת נשחק עם הזמן.
- לעשות את התוכנה קריאה יותר - הקריאות חיונית למתחזקים.
- לעזור למצוא שגיאות - קשה למצוא שגיאה בקוד מסורבל.
- לזרז את כתיבת הקוד - כל השיפורים הללו יקטינו את הזמן שיידרש בהמשך.

מתי לעשות refactoring ?

- כאשר מוסיפים פונקציונליות למערכת - "אם הקוד היה כתוב כך, היה קל יותר להוסיף את הפעולה".
- כאשר צריך למצוא שגיאה - בכל פעם שמסתכלים על קוד ומתקשים להבין אותו יש לבדוק האם ניתן לשפר.
- תוך כדי סקר קוד (Code review)
- באופן כללי, כל פעם שמגלים קוד ש"מריח לא טוב" Bad code smells לדוגמא: כפילות בקוד, שרות ארוך מדי, מחלקה גדולה מדי, רשימת פרמטרים ארוכה, סימפטומים של צימוד חזק מדי בין מחלקות,

קטלוג של refactorings

- הספר של Fowler כולל קטלוג של refactorings שכל אחד כולל שם, סיכום קצר, מוטיבציה, תהליך השינוי, ודוגמא.
- חלק מה refactorings ניתנים לאוטומציה, וכן אקליפס (וגם כלי פיתוח אחרים) תומכים במספר refactorings
- הכלים מאפשרים לראות כיצד ייראה הקוד אחרי השינוי, ולהחליט (וכן לבטל שינוי שנעשה).
- הכלים יכולים לציין מתי מובטח שהשינוי נכון (כלומר לא משנה התנהגות).
- אפילו דוגמא פשוטה - שינוי שם של שרות - קשה מאד לשינוי ידני ללא שגיאה. (שינוי גלובלי בעטרך טקסט לא יהיה נכון בהכרח).

דוגמאות מקטלוג ה refactorings

- extract method / inline method
- Introduce Explaining Variable
- Move method/Field
- Rename method
- Add/Remove Parameter
- Pull up/Push down Field/Method
- Extract Subclass/Superclass/Interface
- Collapse Hierarchy
- Replace Inheritance with Delegation / vice versa

תיכון בעזרת חוזים

- אפיון (חלקי) כחלק מהקוד.
- חלוקת אחריות בין רכיבים במקום תכנות דפנסיבי.
- תנאים (ביטויים בוליאניים):
- תנאי קדם ותנאי אחר של שרות - חוזה בין לקוח לספק
- משתמר של מחלקה - חוזה בין שרותי המחלקה
- תנאי מיוצא (אם כל המשתתפים בביטוי מיוצאים public) לעומת תנאים חסויים.
- תנאי קדם חייב להיות מיוצא כדי שהלקוח יוכל לבדוק.
- תנאי אחר ומשתמר יכולים לכלול חלקים מיוצאים וחלקים חסויים.

תיכון בעזרת חוזים (המשך)

- הפרדה בין פקודה לשאילתא (פרט למקרים יוצאים מן הכלל).
- גישה פרגמטית - אפיון חלקי ולא בהכרח מלא.
- גישה פרקטית - רצוי כלי שיאפשר מעקב בזמן ריצה. לכן ייכתבו תנאים שבדיקתם לוקחת זמן סביר (קבוע).
- הנחה מקובלת - כל פרט שאין אליו התייחסות בתנאי אחר, מניחים שלא משתנה כתוצאה מהפקודה.
- פרסום החוזה - כלי תיעוד (כגון JavaDoc) אמור לאפשר ללקוח לראות רק את החלק המיוצא של החוזה, ולמתחזק את כל החוזה. (לא נסינו לעשות זאת).

תיכון בעזרת חוזים וירושה

- בירושה מתקיים עקרון ההחלפה: אם מחלקה B יורשת ממחלקה A (ישירות או בעקיפין) אזי בכל מקום בו הלקוח מצפה לעצם מטיפוס A, יכול להופיע עצם מטיפוס B.
- לכן אסור למחלקה המרחיבה להחליש את החוזה
- אסור לדרוש תנאי קדם יותר חזק (יותר קשה לקיום)
- אסור להבטיח תנאי אחר יותר חלש
- אסור להודיע על חריג מטיפוס שלא הוצהר במקור (זה נמנע תחבירית בג'אוה)
- המחלקה היורשת (המרחיבה) היא "קבלן משנה".

כתיבת החוזים

- חוזים כהערות מובנות בג'אווה, עם התגיות
pre, post, inv, imp_post, imp_inv
- קשר AND מרומז בין תנאים שונים
- בתנאי האחר, x $prev$ $\$$ מציין ערך של x בכניסה לשרות, ו ret $\$$ מציין את הערך שהשרות מחזיר
- בירושה, לא חוזרים על התנאים של המחלקה המורישה (כנ"ל של מנשק אותו מממשים), אלא ניתן להוסיף להם כך שהחווה של שירות דורס (או של שירות שמממש מנשק אבל עם חווה מחוזק) הוא תמיד מהצורה
- תנאי הקדם הוא "תנאי קדם נורש או תנאי קדם אחר"
- תנאי האחר הוא "תנאי אחר נורש וגם תנאי אחר נוסף"

עקרונות נוספים

- שימוש במנשקים כאשר יש לזה הצדקה (לא לכל מחלקה יש טעם לכתוב מנשק מיוחד עבודה).
- ירושה רק כאשר מתקיים יחס is-a
- האצלה כתחליף לירושה (יחס has-a) - למניעת הכפלת קוד.