חלק 13

תבניות תיכון

(Design Patterns)

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 pattersn 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.

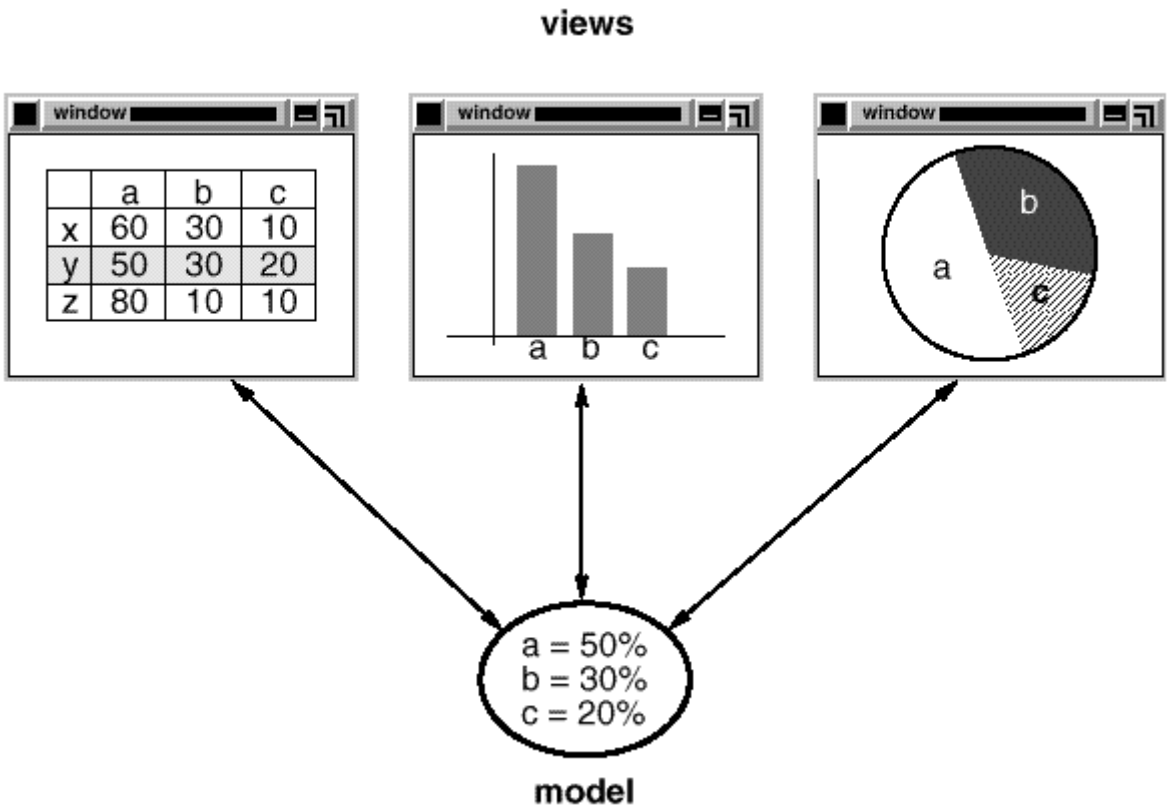
מודל לעומת מראה

- 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

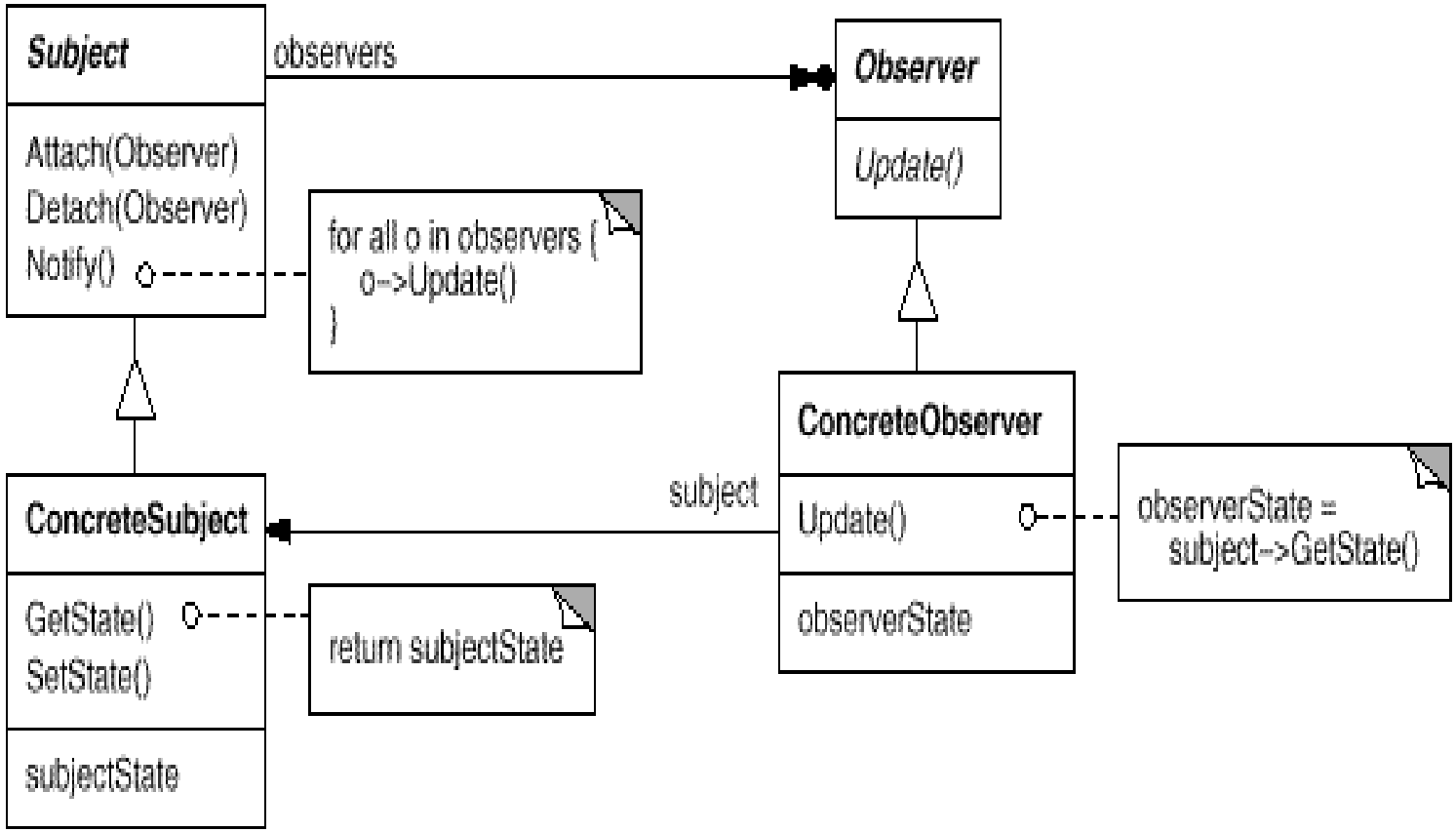
- מטרה: להגדיר תלות של אחד לרבים בין עצמים, כך שכאשר האחד משנה את מצבו, העצמים התלויים מקבלים הודעה ומתעדכנים אוטומטית. (תבנית התנהגות).
- מוטיבציה: לשמור על עקביות בין עצמים קשורים, בלי לגרום לצימוד חזק מדי.
- ישימות:
  - כאשר להפשטה יש שני הבטים, אחד תלוי בשניץ עצמים נפרדים מקלים על שינוי ושימוש חוזר.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

תבנית התיכון Observer - מוטיבציה



תבנית התיכון Observer - מבנה

נשתמש בדיאגרמת מחלקות



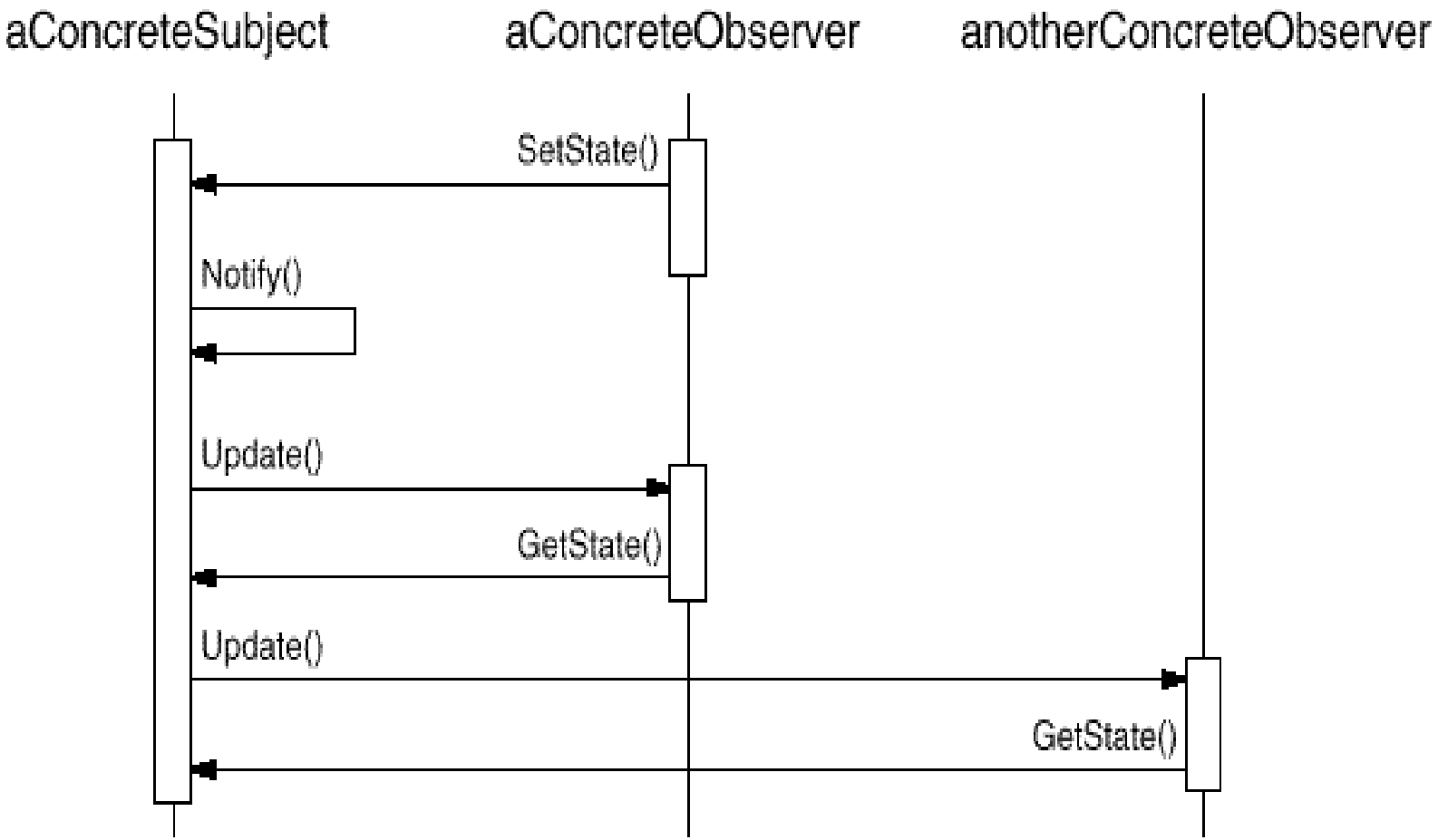
תבנית התיכון Observer - משתתפים

- 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.

<u>הערה</u> (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



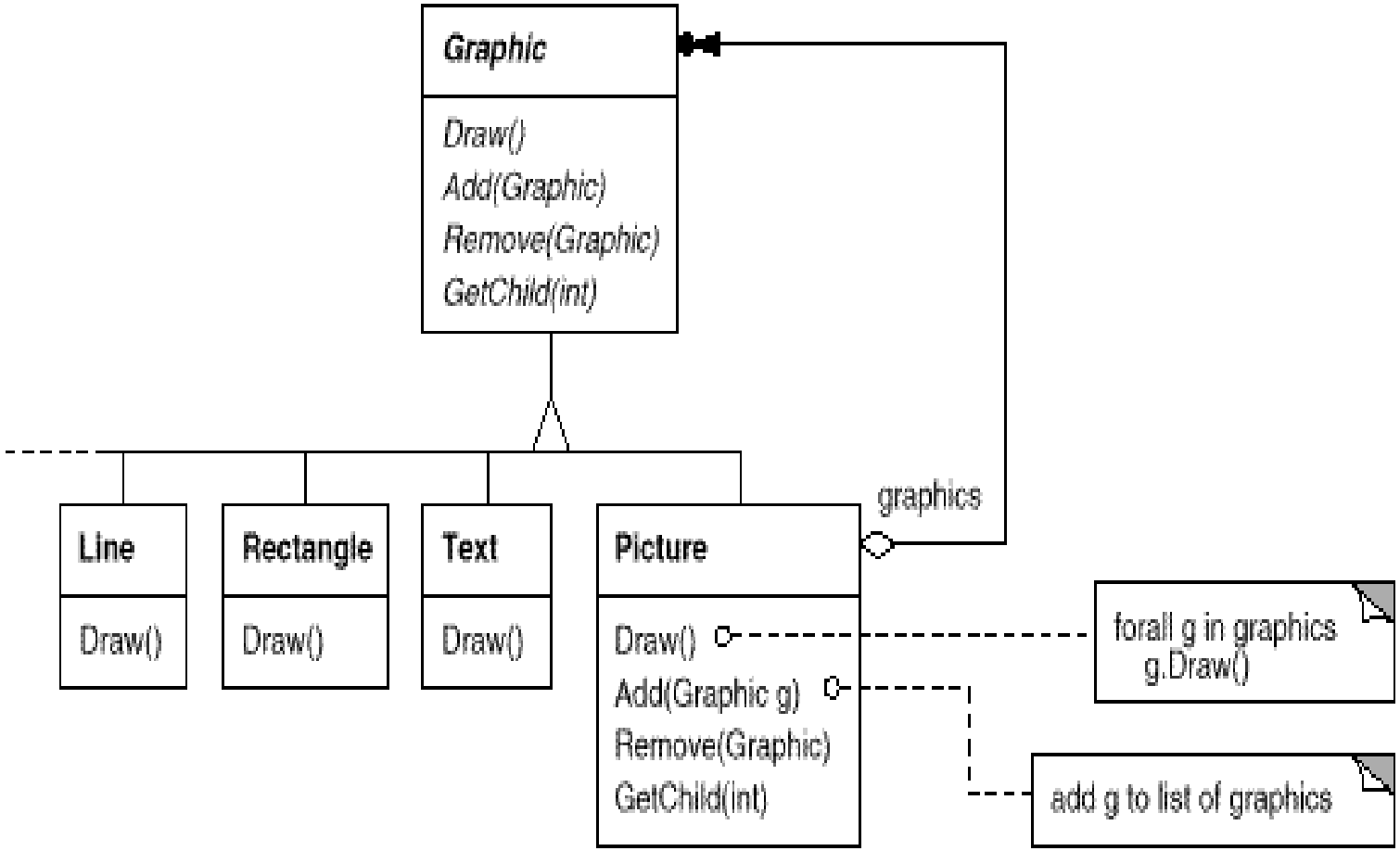
Model View Controller - המשך

- 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

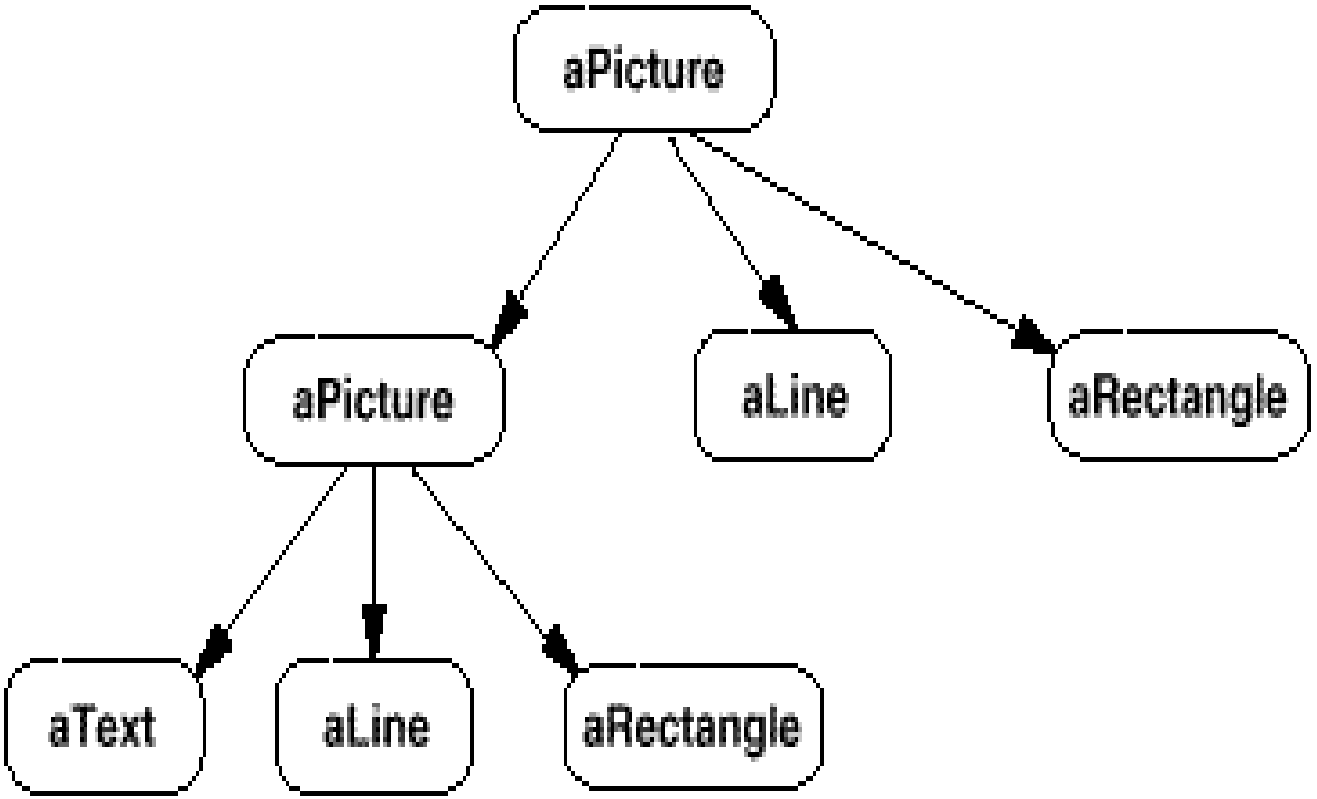
- מטרה: להרכיב עצמים למבני עץ שמייצגים את ההיררכיה של היחס חלק-שלם. לאפשר ללקוחות לטפל בעצמים בודדים ובהרכבות באופן אחיד. (תבנית מבנה).
- דוגמא: עצמים גרפיים.
- ישימות:
  - כאשר רוצים לממש יחס חלק-שלם.
  - כאשר רוצים לאפשר ללקוחות להתעלם מהבדלים בין עצמים מורכבים לעצמים בודדים.

תבנית התיכון Composite - דוגמא

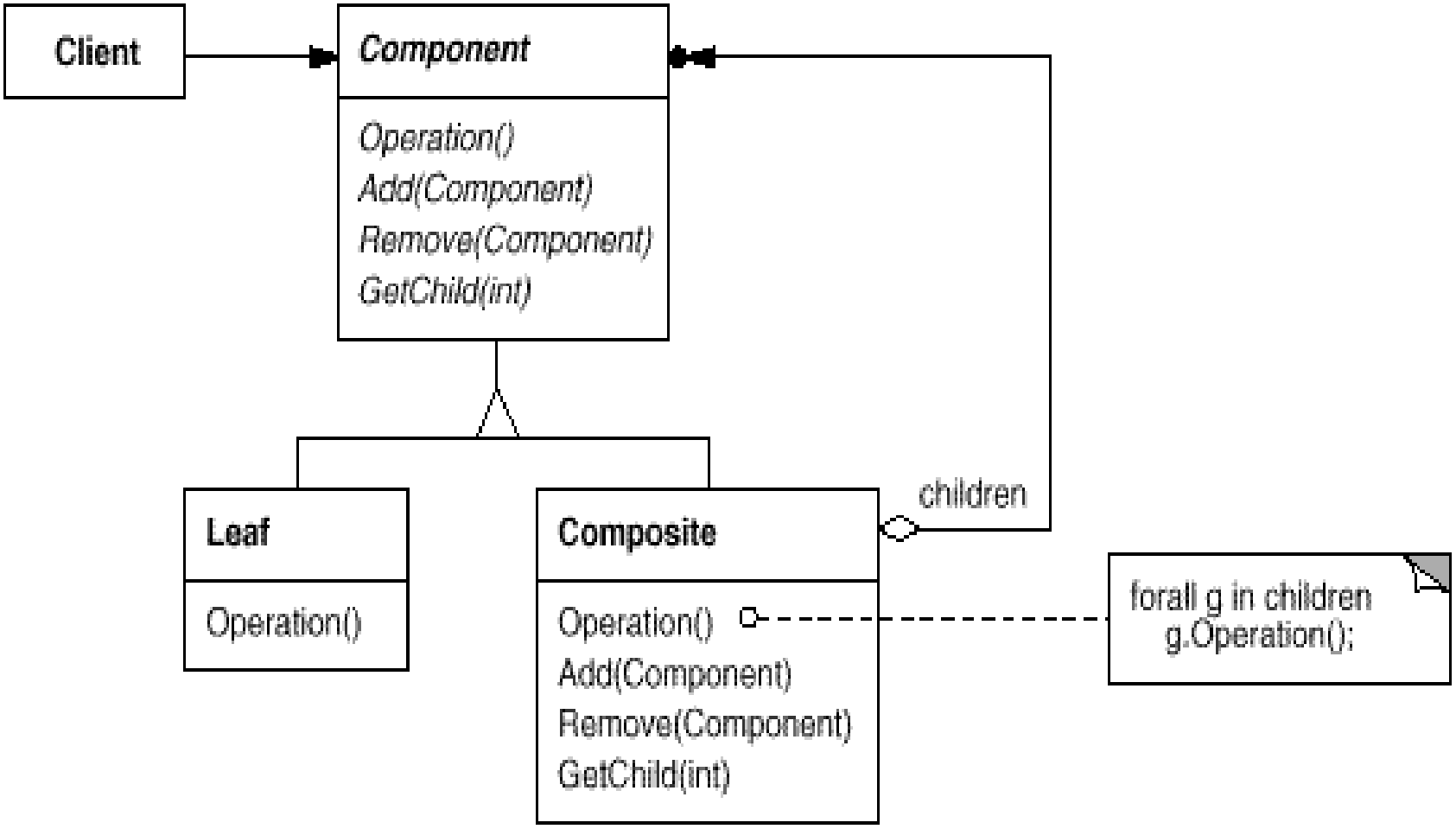


תבנית התיכון Composite - דוגמא

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 .

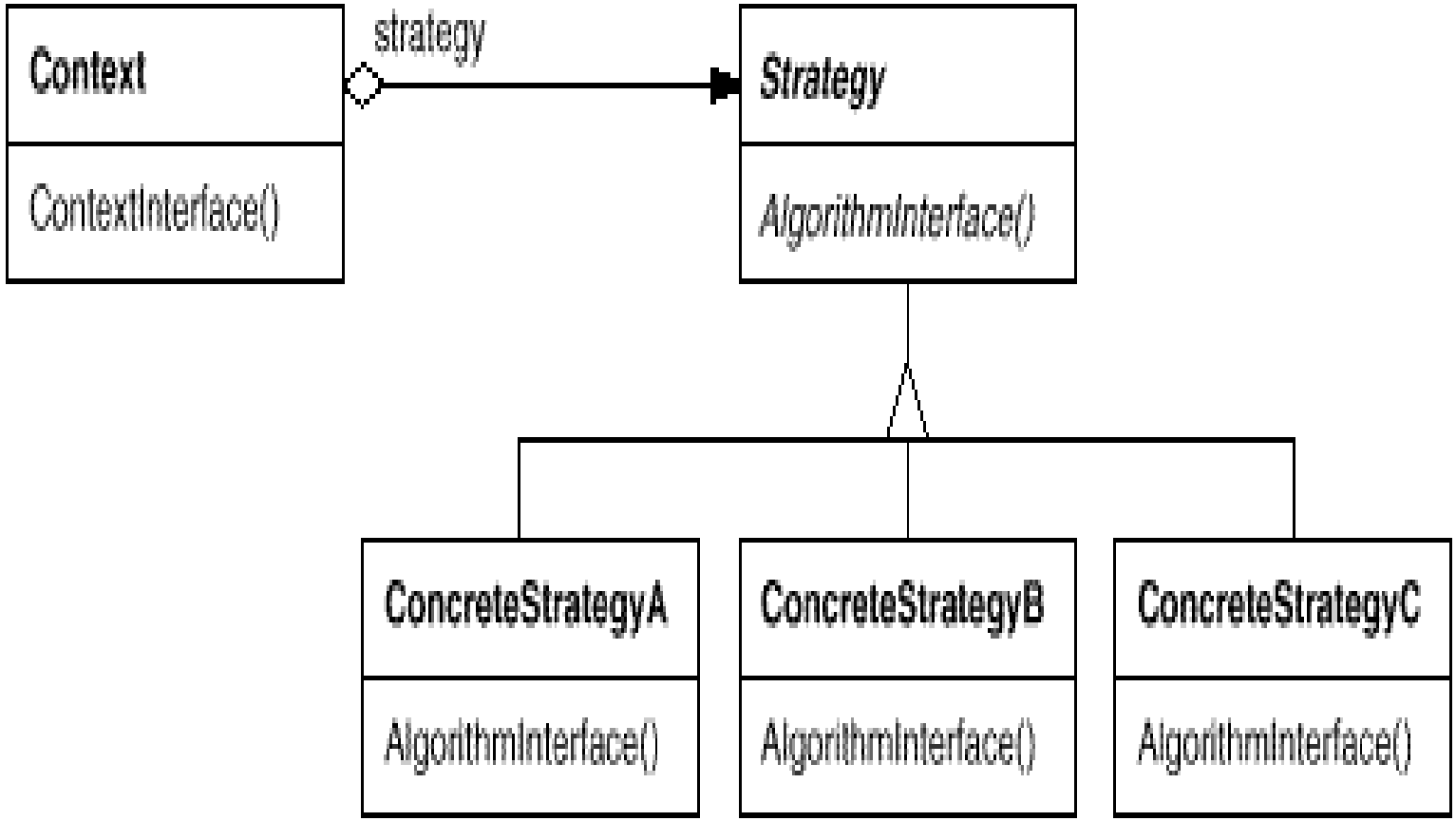
Model View Controller - המשך

- 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 .

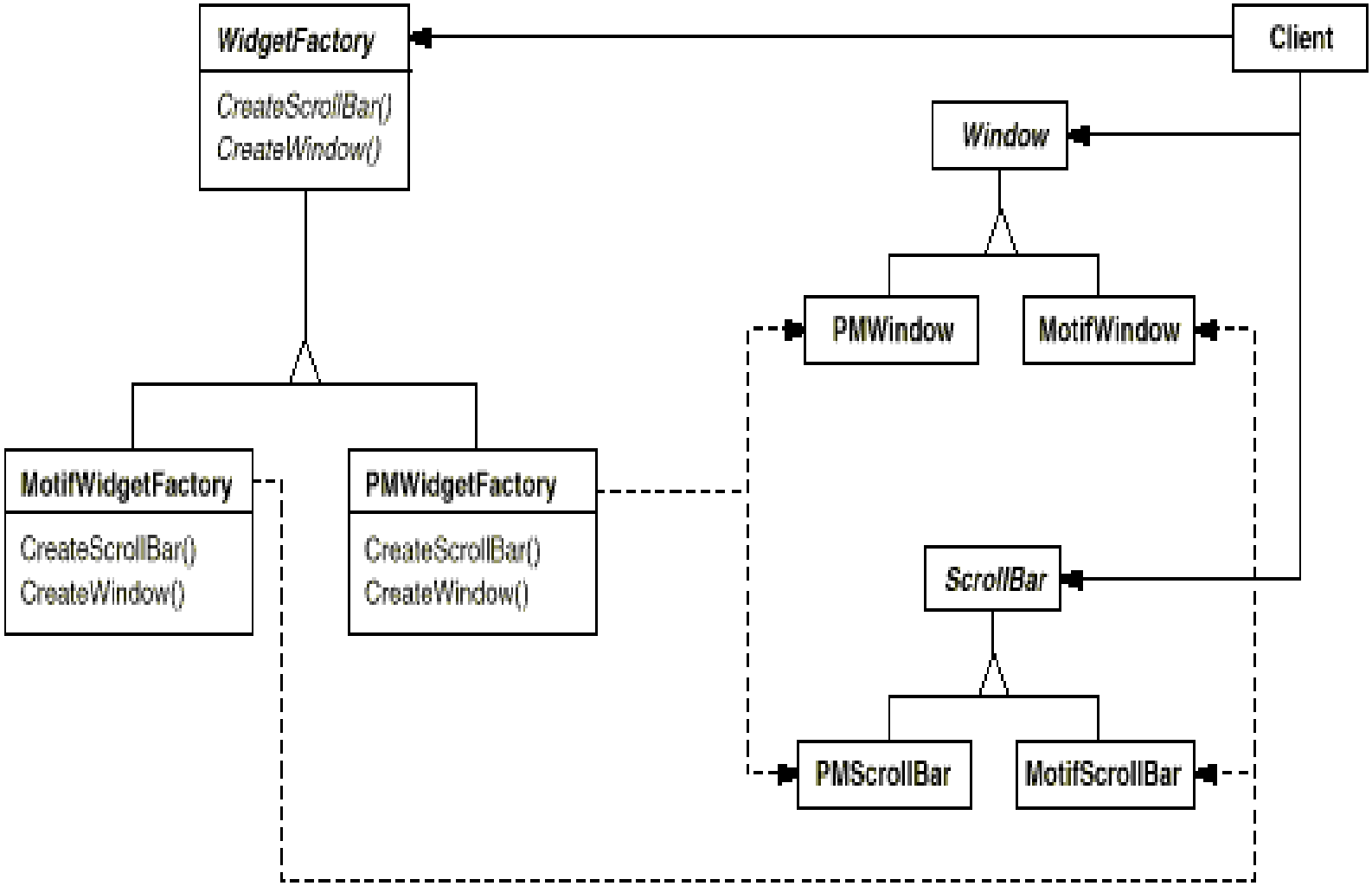
תבניות תיכון ליצירה

- תבנית התיכון Singleton נועדה להבטיח שיש רק אוביקט אחד ממחלקה מסוימת. ראינו איך ליישם זאת בג׳אוה.
- דברנו גם על בית חרושת, גם זאת תבנית תיכון ליצירה.
- למעשה יש כמה תבניות תיכון העוסקות בבית חרושת. נציג עכשיו מקרה מורכב יותר - תבנית התיכון Abstract Factory.

תבנית התיכון Abstract Factory

- מטרה: לספק מנשק ליצירת משפחה של עצמים קשורים, בלי לקבוע את המחלקות המוחשיות.
- דוגמא: יישום מבוסס חלונות שצריך לרוץ על מספר מערכות חלונות תוך תלות מינימלית במערכת החלונות.
- כללית - יש מספר מוצרים ומספר גירסאות (או משפחות)ץ כל מוצר זמין בכל המשפחות. המערכת משתמשת במוצרים ממשפחה אחת, אך התלות במשפחה צריכה להיות מזערית.

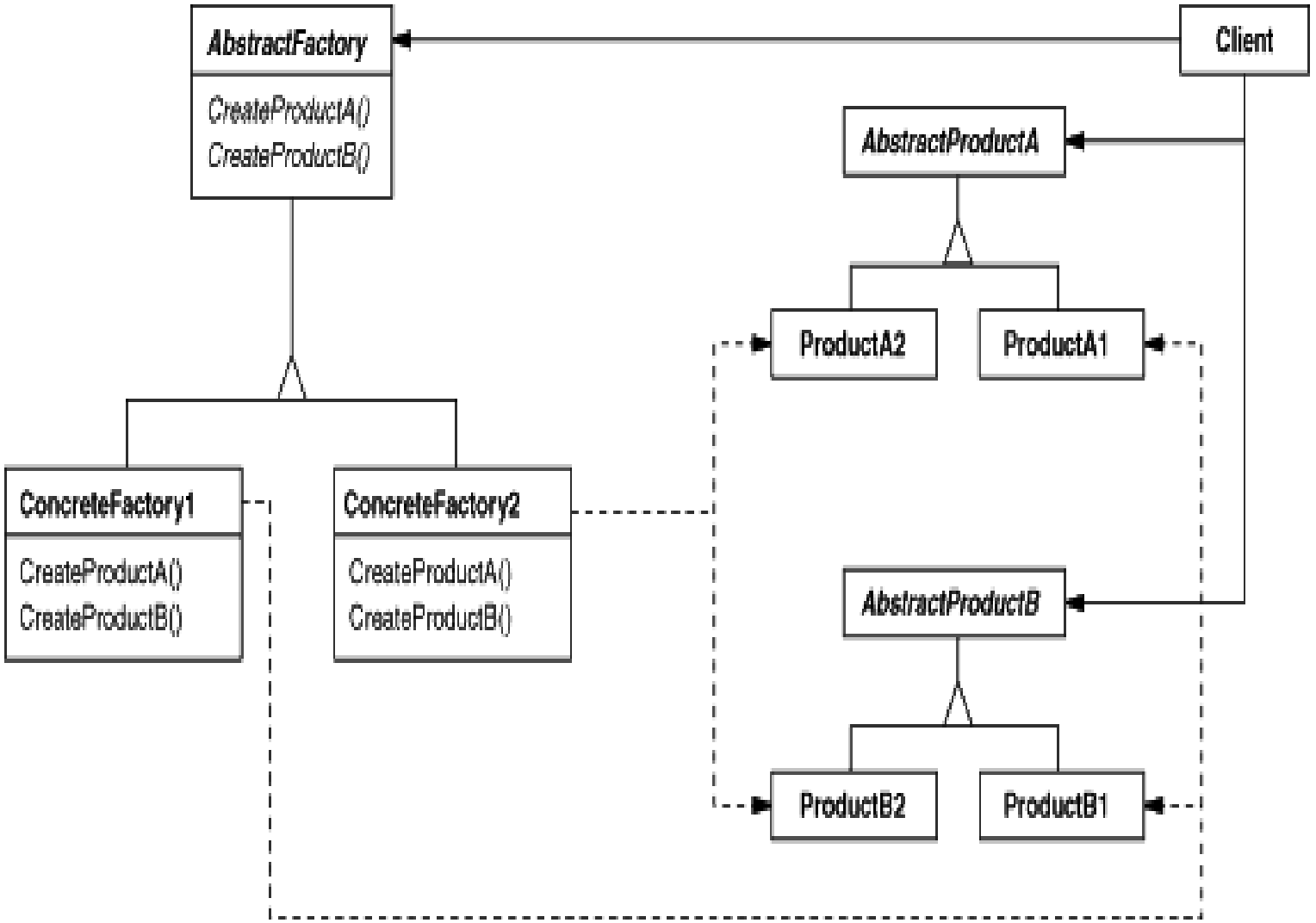
תבנית תיכון Abstract Factory - דוגמא



תבנית תיכון Abstract Factory - ישימות

- כאשר מערכת צריכה להיות תלויה באופן שבו מוצרים נוצרים, מורכבים ומיוצגים.
- כאשר המערכת צריכה להיות מקונפגת לאחת מבין מספר משפחות של מוצרים.
- כאשר משפחה של מוצרים מתוכננת לעבוד ביחד, ועלינו לאכוף זאת.
- כאשר רוצים לספק ספריה של מוצרים, ולגלות רק את המנשקים.

תבנית תיכון Abstract Factory - מבנה



תבנית תיכון Abstract Factory - משתתפים

- AbstractFactory - מגדיר שרותים מופשטים ליצירת כל נמוצרים המופשטים.
- ConcreteFactory |(N גרסאות) - יורש מ AbstractFactory ומיישם את השרותים ליצירת מוצרים מוחשיים.
- AbstractProduct (M גרסאות) - מגדיר מנשק לטיפוס של מוצר.
- ConcrteteProduct (N*M גרסאות) חורש מ AbstractProduct מסוים, ומיישם את הממשק. מיועד להווצר ע"י ConcreteFactory מסוים.

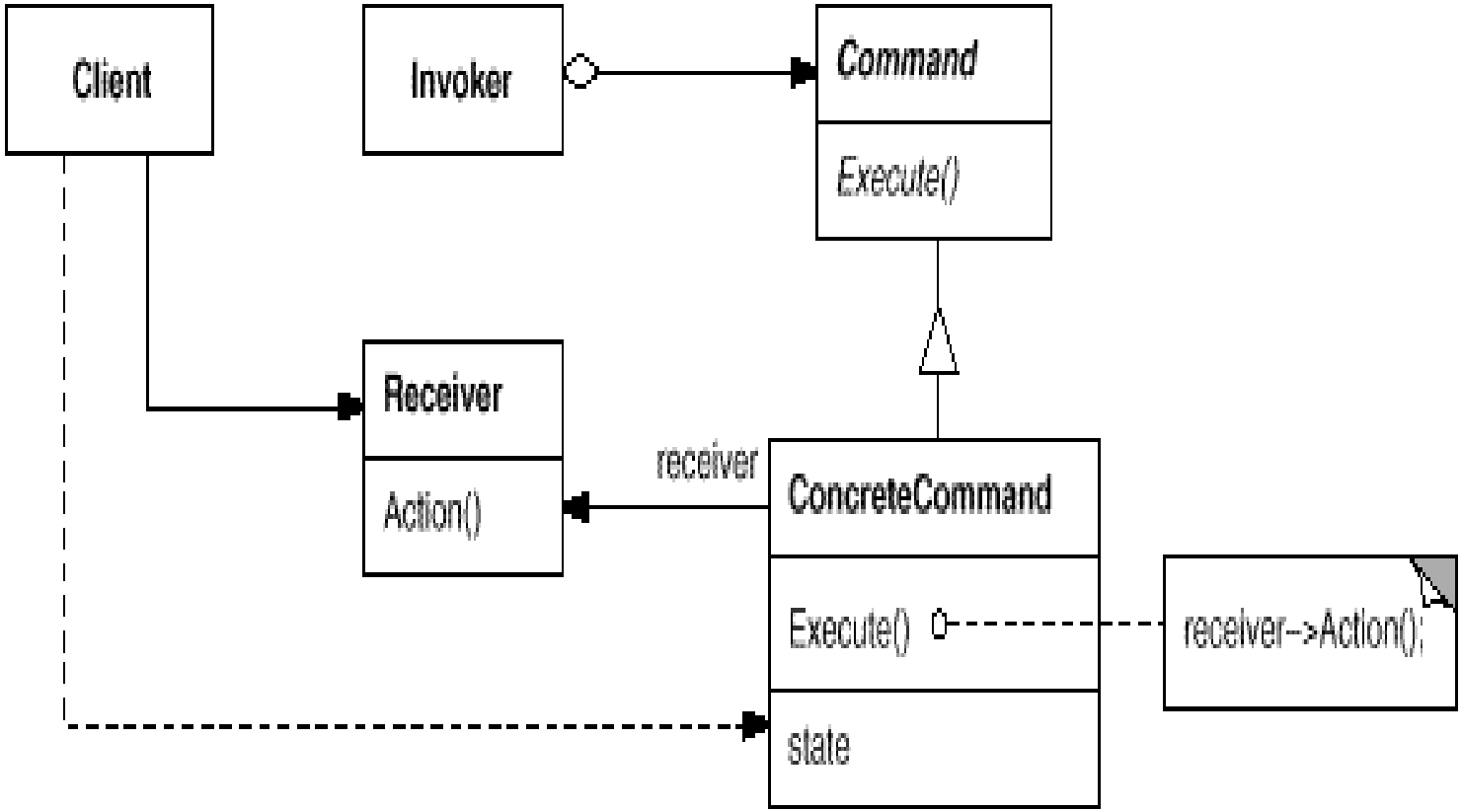
תבנית תיכון Abstract Factory - משתתפים (המשך)

- Client - לקוח של AbstractFactory ושל מספר AbstractProduct . שתי אפשרויות:
- משתמש ב AbstractProduct אחד בלבד, וכדי להחליפו יש לשנות מקום אחד בקוד ולקמפל.
- תומך ביותר מ AbstractProduct אחד. משתמש בפיצול (משפט switch) במקום אחד לכל היותר ביצירת עצם בית החרושת.

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. מממש 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 להוות 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׳אוה).
- 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.

חלק 14

כיצד פועלים

מנגנוני השפה

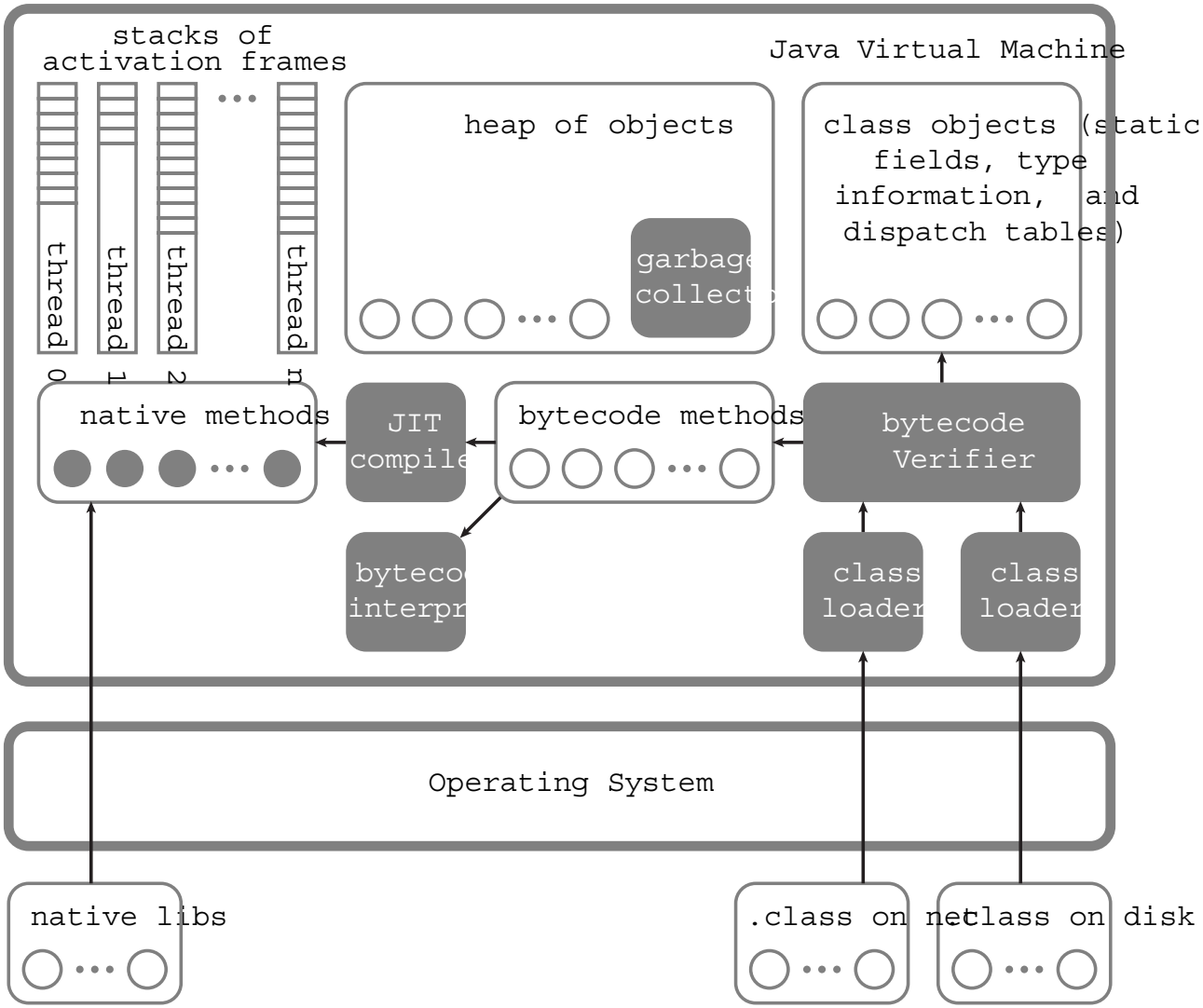
מבט מלמעלה

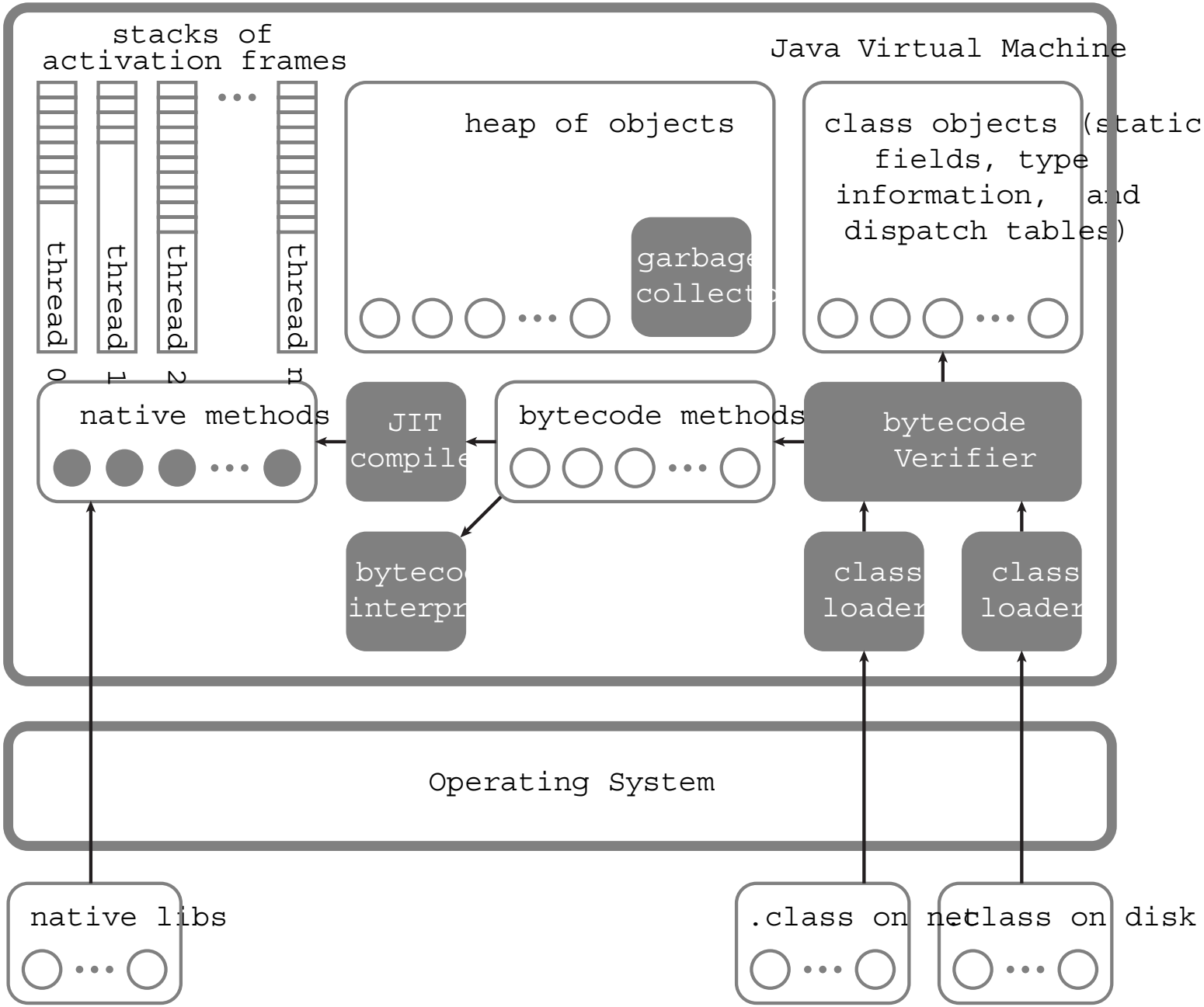
- הקומפיילר מתרגם את קוד המקור ל-byte code, קובץ `.class`, שמכיל ייצוג בינארי דחוס של מחלקה; לא נדון כמעט בשלב הזה
- הקובץ הבינארי משקף כמעט בדיוק את מבנה הקוד
- כדי להריץ תוכנית ג׳אווה, מערכת ההפעלה מפעילה תוכנית ״ילידה״ (כתובה בדרך כלל בשפת C ומשתמשת ישירות במנשקים של מערכת ההפעלה) בשם `java`
- התוכנית הזו היא Java Virtual Machine (JVM) והיא טוענת קבצים בינאריים, בדידים או במארז `.jar`, ומבצעת את הפקודות שהם מכילים, כולל הקצאת ושחרור זיכרון וקריאה לשירותים

קומפילציה

- מה הקומפיילר צריך לדעת כאשר הוא מקמפל מחלקה?
- הקומפיילר צריך לדעת מהם השדות והשירותים שמוגדרים בכל טיפוס שהמחלקה משתמשת בו בשדות, משתנים, וארגומנטים
- מהיכן הקומפיילר שואב את המידע הזה?
- בדרך כלל, מהקבצים הבינאריים של הטיפוסים הללו
- אבל אם הם עדיין לא עברו קומפילציה, הקומפיילר מחפש את קוד המקור ומקמפל אותם ביחד עם המחלקה הנוכחית
- אין הפרדה בין קובץ שמכיל רק הצהרות על השדות והשירותים ובין קובץ שמכיל את ההגדרות שלהם, כמו שיש בשפות אחרות (למשל C++), ולכן אין סיכוי שההצהרות וההגדרות לא יתאימו אלה לאלה

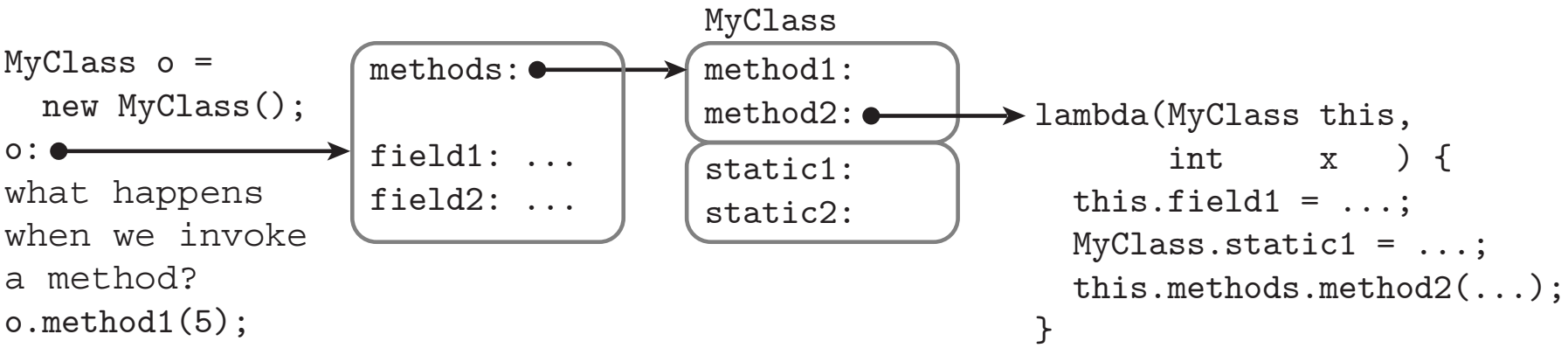
ומכאן, לעולם של המכונה הוירטואלית




stacks of
activation frames
Java Virtual Machine
thread 0
thread 1
thread 2
thread n
heap of objects
garbage
collect
class objects (static
fields, type
information, and
dispatch tables)
native methods
JIT
compile
bytecode methods
bytecode
Verifier
byteco
interpr
class
loader
class
loader
Operating System
native libs
.class on net
class on disk

מבנה של עצם

```
class MyClass {
 static int static1,static2;
        int field1, field2 ;
 void method1(int x) { field1=...; static2=...; method2(...); }
 void method2(int y) {...}
 ...
}
```

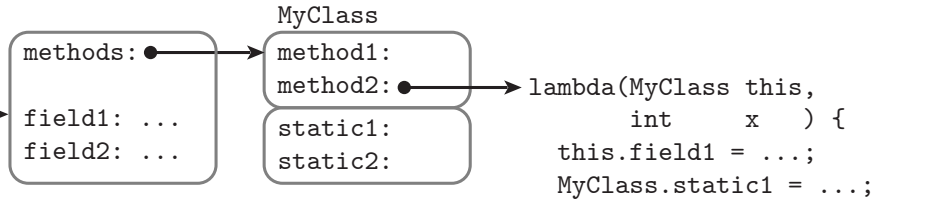


מבנה של עצם ושל ייצוג של מחלקה

- בזמן ריצה, עצם הוא מבנה נתונים שמכיל הצבעה למבנה הנתונים של המחלקה שהוא שייך אליה ואת הערכים של שדות המופע
- התייחסות היא הצבעה למקום בזיכרון שבו מתחיל המבנה של העצם המיוחס (לפעמים יותר מסובך; נסביר בהמשך)
- הייצוג של מחלקה כולל מידע על הטיפוס (בעיקר איזה מנשקים היא מממשת), טבלת הצבעות לשירותים (dispatch table), ואת הערכים של שדות המחלקה

```
class MyClass {
 static int static1,static2;
        int field1, field2 ;
 void method1(int x) { field1=...; static2=...; method2(...); }
 void method2(int y) {...}
 ...
}

MyClass o =
  new MyClass();
o:
what happens
when we invoke
a method?
o.method1(5);
```




```
lambda(MyClass this,
       int     x   ) {
  this.field1 = ...;
  MyClass.static1 = ...;
  this.methods.method2(...);
}
```

ייצוג של שירות

- שירות מופע (לא `static`) הוא שגרה שמקבלת את הארגומנטים הפורמאליים של השירות, וגם ארגומנט נסתר שבו מועברת הכתובת בזיכרון של העצם (`this`)
- טבלת ההצבעות לשירותים של מחלקה יכולה להצביע לשירותים שהגדירה וגם לשירותים שירשה ולא דרסה; לכן, שירות מופע `m` צריך להיות מוכן לקבל `this` שמצביע לעצם שאינו מהמחלקה שהגדירה את `m` אלא ממחלקה מרחיבה
- שירות מחלקה (`static`) הוא שגרה שמקבלת את הארגומנטים הפורמאליים של השירות, ולא מקבלת מצביע ל-`this`
- פרט לכך שירותי מופע ומחלקה זהים, ושני הסוגים מופיעים באותה טבלת ההצבעות לשירותים של מחלקה

שימוש בשדות

- שימוש בשדה מופע מתבצע על ידי הוספת ההיסט של השדה לכתובת `this`; הכתובת שמתקבלת היא הכתובת של השדה
- בדוגמה השתמשנו בסימון `this.field1`
- שימוש בשדה מחלקה יותר פשוט; כאשר המחלקה נטענת לזיכרון, נקבעות הכתובות של שדות המחלקה, שלא זזים במהלך התוכנית; אפשר להחליף כל התייחסות לשדה מופע בהתייחסות לכתובת של השדה בזיכרון
- בדוגמה הסימון היה `MyClass.static1`, אבל בעצם זו כתובת אבסולוטית

הפעלה של שירות

- למשל `o.method1(5)`
- ההתייחסות `o` מצביעה למקום של עצם בזיכרון
- המבנה של העצם כולל הצבעה לטבלת השירותים שלו (השדה הנסתר `methods`)
- השירות `method1` הוא השירות הראשון של המחלקה, ולכן ההצבעה לשגרה תהיה במקום הראשון בטבלת השירותים
- את השגרה הזו מפעילים, כאשר בארגומנט הראשון (הנסתר) שלה תועבר הכתובת של `o` ובארגומנט השני הערך 5
- בסימונים שלנו, זה `o.methods[0] ( o, 5 )`
- להפעלה כזו של שירות קוראים הפעלה וירטואלית, מכיוון שהשירות שיופעל תלוי בטיפוס הדינאמי, לא הסטאטי

אופטימיזציה: devirtualization

- אם ברור שהטיפוס הדינאמי של o זהה לטיפוס הסטאטי שלו, אז אין צורך בהפעלה וירטואלית
- למשל, בקוד

```
MyClass o = new MyClass();
o.method1(5);          clearly o is a member of MyClass
```

- או אם `MyClass` מוגדר `final` או שהשירות `method1` מוגדר במחלקה `final`; זה מונע דריסה שלו
- במקרים כאלה, בזמן הטעינה של המחלקה שקוראת לשירות אפשר להחליף את ההפעלה הוירטואלית בהפעלה של השגרה המסוימת שצריך להפעיל על פי כתובתה בזיכרון
- אין צורך בחישוב הכתובת בעזרת ביטוי כמו `o.methods[0]`

מבנה עצם ממחלקה מרחיבה

- כאשר מחלקה Sub מרחיבה את המחלקה Base, המבנה של עצמים מהמחלקה המרחיבה נגזר ממבנה עצמים ממחלקת הבסיס
- גם המבנה של ייצוג המחלקה עצמה נגזר מייצוג הבסיס
- שדות מופע בעצמים של Sub יופיעו לאחר שדות המופע שהוגדרו כבר ב-Base
- שירותים שנוספו ב-Sub יופיעו לאחר השירותים שנורשו או נדרסו מ-Base
- זה מבטיח שהתייחסות לעצם דרך מצביע מטיפוס Base תפעל נכון: השדות והשירותים נמצאים באותו מקום יחסי ב-Sub וב-Base

הקושי בהפעלת שירותים על מנשקים

- כאשר מחלקה Sub מרחיבה את Base, שמרחיבה, למשל, את Object, אז השירותים והשדות של Object הם תת קבוצה של אלה של Base שהם תת קבוצה של אלה של Sub
- זה מאפשר לסדר את טבלת השירותים ואת השדות כך שתתי הקבוצות יופיעו תמיד כתחיליות; אפשר להשתמש בעצם דרך כל אחד משלושת הטיפוסים
- המצב יותר מסובך אם Sub מממשת שני מנשקים, I1 ו-I2
- אין קשר בין השירותים ששני המנשקים מצהירים עליהם
- אי אפשר לסדר את השירותים כך שאפשר יהיה למצוא את השירותים של I1 ואת השירותים של I2 באותה טבלת שירותים (dispatch table) בלי להתייחס לטיפוס הדינאמי
- בהפעלה o.methods[0] לא התייחסנו לטיפוס הדינאמי

הפעלת שירות על מנשק

- אז איך מפעילים את השירות `m` על עצם `o` שטיפוס הסטאטי שלו הוא המנשק `I1`?
- בעיה דומה יש בשפות כמו C++ ו-Eiffel שמתירות ירושה מרובה
- זו בעיה קשה שנמצאת עדיין בחזית המחקר
- מימושים גרועים של ג׳אווה משתמשים באלגוריתם פשוט שמחפש את השירות הנחוץ; הפעלה כזו איטית בסדר גודל אחד או שניים מהפעלה וירטואלית
- במימושים מתוחכמים אין כמעט הבדל ביצועים בין הפעלה וירטואלית ובין הפעלה של מנשק, אבל הם מסובכים למדי או בזבזניים בזיכרון
- אנו נציג פתרון יעיל ופשוט, אבל אופייני דווקא ב-C++

הפעלה יעילה של שירות על מנשק

- עבור כל מחלקה, נשמור לא טבלת שירותים (dispatch table) אחת, אלא מערך שלם של טבלאות כאלה, אחת עבור כל טיפוס שהמחלקה מתאימה לו
- ליתר דיוק, צריך במערך טבלה אחת עבור הפעלות וירטואליות ועוד טבלה אחת עבור כל מנשק שהמחלקה מממשת ושמרחיב יותר ממנשק אחד
- התייחסות לעצם תכלול גם מצביע לייצוג של העצם (לשדות שלו) וגם את הכתובת של האיבר במערך טבלאות השירותים שמתאים לטיפוס הסטאטי של התייחסות
- יציקה למעלה או למטה בייצוג כזה דורשת הזזה של הכתובת של האיבר במערך טבלאות השירותים
- הקומפיילר יודע בדיוק בכמה צריך להזיז ביציקה למטה

הרצה וקומפילציה של bytecode

- בקובץ class. השירותים מיוצגים ב-bytecode, שפת מכונה של מחשב וירטואלי (לא כל הפקודות פשוטות, למשל `invokeinterface`)
- לאחר טעינה של מחלקה לזיכרון, ה-JVM יכול להריץ שירותים על ידי סימולציה של המחשב הוירטואלי; הרכיב של ה-JVM שמבצע את הסימולציה נקרא bytecode interpreter
- בסימולציה כזו יש מחיר גבוה גם לפעולות מאוד פשוטות, למשל חיבור של שני שלמים
- כדי להימנע מתקורה קבועה על כל פעולה, תקורה שנובעת מהסימולציה, ה-JVM יכול לקמפל את הקוד של שירות לשפת מכונה של המעבד שהתוכנית רצה עליו

Just-in-Time Compilation

- קומפילציה כזו מ-bytecode לשפת מכונה (native code) נקראת just-in-time compilation, מכיוון שהקומפילציה מתבצעת ממש לפני השימוש בקוד, ולא כצעד מכין לפני אריזת התוכנה להפצה
- בדרך כלל, JIT מופעל על שירות לאחר שהתברר ל-JVM שהשירות מופעל הרבה; זה מונע קומפילציה יקרה של שירותים שאינם מופעלים או כמעט ואינם מופעלים
- אבל יתכנו גם אסטרטגיות אחרות: לעולם לא לבצע JIT, לבצע באופן מיידי בזמן הטעינה של מחלקה, לבצע באופן מיידי אבל ללא אופטימיזציות ולשפר אחר כך את הקומפילציה של שירותים שנקראים הרבה, לבצע באופן גורף אבל רק כאשר המעבד נח והמחשב מחובר לחשמל, ועוד
- עם JIT, הביצועים של תוכנית משתפרים לאורך הריצה

אז למה bytecode?

- אם ממילא תוכנית הג׳אווה מתקמפלת בסופו של דבר לשפת המכונה של המעבד, למה לא לקמפל אותה לשפת מכונה בזמן אריזת התוכנה להפצה, ולא בזמן ריצה?
- קומפילציה בזמן אריזה יעילה יותר, מכיוון שהיא מתבצעת פעם אחת עבור הרצות רבות; בזמן אריזה כדאי להפעיל אופטימיזציות יקרות, בזמן ריצה לא
- הפצת תוכנה כ-bytecode משיגה שתי מטרות; האחת, את היכולת להשתמש בתוכנה ארוזה אחת על מעבדים שונים (ומערכות הפעלה שונות)
- המטרה השנייה היא בטיחות; ה-bytecode verifier בודק את התוכנית לפני שמריצים אותה ומוודא שהיא מקיימת את דרישות השפה; זה משפר את בטיחות מערכות המחשב ומונע סוגים מסוימים של תקיפות

תוכנה לשימוש במספר פלטפורמות שונות

- היכולת לארוז תוכנה פעם אחת ולהשתמש בה במחשבים עם מגוון של מעבדים ומערכות הפעלה היא יכולת חשובה, מכיוון שאריזת תוכנה להפצה היא פעולה מורכבת ויקרה.
- ללא bytecode, צריך לקמפל ולארוז את התוכנה עבור כל פלטפורמה בנפרד. למשל: חלונות על אינטל, חלונות על מעבד אחר (למשל מחשב כף יד או טלפון), מקינטוש, ...
- סיסמת השיווק של סאן לגבי ג׳אוה הייתה " write once run anywhere": לכתוב ולארוז את התוכנה פעם אחת ולהריץ אותה על פלטפורמות רבות.

תוכנה לשימוש במספר פלטפורמות שונות

שתי רמות יכולת של תוכנה לרוץ על מספר פלטפורמות:

- ברמה הראשונה: קוד המקור מתאים למספר פלטפורמות, אבל צריך לקמפל אותן לכל פלטפורמה בנפרד.
- ברמה השנייה: התאמה לא רק ברמת קוד המקור, אלא גם ברמת התוכנה הארוזה להפצה. תוכניות ג׳אווה שייכות לרמה הזו, הודות לשימוש ב-bytecode.

שפת מכונה וירטואלית כמנגנון להפצת תוכנה

הרעיון אינו חדש. (גרסאות מסוימות של פסקל השתמשו בו לפני שנים רבות, למשל).

חסרונות שמנעו שימוש נרחב בעבר:

- ביצועים: פרשן (bytecode interpreter) הוא איטי לעומת ביצוע קוד דומה בשפת מכונה. לכן, שימוש ב-bytecode דורש אחד משני תנאים (או את שניהם): מעבד מהיר מאוד, או קומפילציה בזמן ריצה (JIT).
- מגוון קטן יחסית של מעבדים. הגידול במגוון המעבדים (ומערכות ההפעלה) הביא אפילו את מיקרוסופט, לחפש דרכי הפצה כאלה, והוביל לארכיטקטורת ה-.NET, משפחה של שפות תכנות וסביבת זמן ריצה דומות לג׳אווה.

הטמנת תרגומים לשפת מכונה

- (מנגנון שלא קיים עדיין)
- באופן עקרוני, אין סיבה לבצע JIT בכל ריצה של התוכנית
- ה-JVM (או מערכת ההפעלה במקרה של .NET) יכולה להטמין תרגומים של bytecode לשפת מכונה ולהשתמש בהם שוב ושוב
- אם התרגומים לשפת מכונה נשמרים בקבצים שרק ל-JVM יש אליהם גישה (ואולי חתומים דיגיטאלית), בדיקת התקינות שבוצעה נשארת תקפה ואפשר להשתמש בהם ללא חשש
- אפשר גם לבצע קומפילציה עם אופטימיזציות יקרות כשהמחשב אינו פעיל או בזמן התקנת התוכנה
- כבר היום יש מערכות הפעלה שמבצעות אופטימיזציות מסוימות בזמן התקנת תוכנה (תוכנה ילידה)

אופטימיצזיות בזמן התקנה

- מערכת ההפעלה של המקינטוש, למשל, מבצעת אופטימיזציה מסוימת בזמן התקנת תוכנה. האופטימיזציה הזו, שנקראת prebinding, מתבצעת על תוכניות שכבר קומפלו לשפת מכונה, והיא מיועדת להאיץ את ההפעלה שלהן.

איסוף זבל (garbage collection)

- מנגנון אוטומטי לזיהוי עצמים ומערכים שהתוכנית לא יכולה להגיע אליהם יותר ושחרור הזיכרון שהם תופסים

```
Double w = new Double(2.0);
Double x = new Double(3.0);
Double y = new Double(4.0);
Double z = new Double(5.0);
w.compareTo(x);
y = null;
```

Can we now release w, x, y, z?

- קשה לדעת; `compareTo` הייתה עשויה לשמור במבנה נתונים כלשהו התייחסויות ל-`w` ו-`x`; ברור שאפשר לשחרר את `y` , אבל ב-`z` השירות עוד עשוי להשתמש

מהו זבל?
לאיזה עצמים אי אפשר להתייחס?

- יותר קל להגדיר את העצמים שאליהם כן אפשר להתייחס
- ראשית, לעצמים שיש אליהם התייחסות ממשתנים אוטומטיים של גושי פסוקים שלא סיימו לפעול, כלומר שגרות וגושי פסוקים פנימיים שפעולתם הופסקה בגלל הפעלה של שירות או פרוצדורה או בגלל גוש פנימי יותר
- שנית, לעצמים שיש אליהם התייחסות משם גלובאלי; בג׳אווה, שמות גלובליים מתאימים בדיוק לשדות מחלקה
- ושלישית, לכל עצם שיש אליו התייחסות מעצם שכבר הוּכח שניתן להתייחס אליו; זו הגדרה רקורסיבית, אבל היא היחידה הנכונה
- כל השאר זבל

שורשים

- תהליך איסוף זבל מתחיל בשורשים, התייחסויות שברור שלתוכנית יש גישה אליהם
- שורשים בג׳אווה כוללים שדות מחלקה ואת כל המשתנים שנמצאים בחלק החי של כל המחסניות (אחת אם יש רק חוט/תהליכון אחד בתוכנית, יותר אם היא מרובת חוטים)
- אם נסמן את השורשים בצבע מיוחד, ואחר כך נצבע באותו צבע כל עצם לא צבוע שיש אליו התייחסות מעצם צבוע, ונמשיך עד שלא יהיה מה לצבוע, העצמים הצבועים אינם זבל וכל השאר זבל
- זו תמיד ההגדרה של זבל; יש אלגוריתמים לאיסוף זבל שפועלים ממש בצורה הזו (mark and sweep), ויש שפועלים בצורה אחרת

איסוף זבל בשיטת mark & sweep

- אוסף הזבל עוצר את התוכנית
- עוברים על כל העצמים והמערכים שנגישים (reachable) מהשורשים, ומסמנים אותם (צובעים אותם)
- עוברים על כל העצמים, ומשחררים את הלא מסומנים; הזיכרון שתפסו יוקצה בהמשך לעצמים אחרים
- אבל יש עוד גישות

איסוף זבל בשיטת ההעתקה (copying)

- הזיכרון מחולק לשני חלקים באותו גודל, א׳ וב׳
- בזמן שהתוכנית פועלת, כל העצמים והמערכים נמצאים בצד אחד; הצד השני ריק
- אוסף הזבל עוצר את התוכנית; נניח שכל העצמים בצד א׳
- עוברים על כל העצמים והמערכים שנגישים מהשורשים, ומעתיקים כל אחד מהם לצד ב׳
- המיקום של עצמים בזיכרון משתנה; צריך לעדכן התייחסויות אליהם; אפשר לעשות זאת על ידי סימון עצמים בא׳ שמועתקים כלא תקפים וסימון המקום החדש שלהם
- כאשר לא נשארים עצמים נגישים בצד א׳, מוחקים את כולו
- באיסוף הבא התפקידים של א׳ וב׳ מתחלפים

עדכון התייחסויות באוסף מעתיק

- נניח שמעתיקים עצם מכתובת p1 בזיכרון בצד א׳ לכתובת p2 בצד ב׳
- משנים את תוכן שטח הזיכרון בכתובת p1
- מדליקים סיבית שמסמנת שהעצם כבר הועתק לצד ב׳
- כותבים בשטח הזיכרון את הכתובת החדשה p2
- לאחר סיום ההעתקה, עוברים על כל העצמים בצד ב׳
- עבור כל עצם, מוצאים את כל ההתייחסויות שהוא מכיל (שדות מופע שהם התייחסויות ומערכים של התייחסויות)
- עבור כל התייחסות כזו לעצם בכתובת q1 שולפים מהעצם ב-q1 את הכתובת החדשה של העצם q2 ומעדכנים
- מעדכנים באופן דומה את השורשים

איזו גישה עדיפה?

- יש הבדלי ביצועים משמעותיים בין mark & sweep ובין copying collectors
- מה ההבדלים ואיזו גישה עדיפה?

השוואת שתי הגישות לאיסוף זבל

- זמן הריצה של mark & sweep תלוי במספר העצמים בזיכרון בזמן האיסוף
- טיפול בכל עצם דורש מספר קבוע של פעולות; סימון של עצמים נגישים ושחרור עצמים לא נגישים
- זמן הריצה של copying collectors תלוי בכמות הזיכרון הכוללת שתופסים העצמים הנגישים, כי צריך להעתיק אותם
- אבל אין שום טיפול בעצמים לא נגישים; הם נמחקים בבת אחת

ספירת התייחסויות

- גישה נוספת לאיסוף זבל, שלא עובדת בג׳אווה, מבוססת על ספירת ההתייחסויות לכל עצם/מערך
- בכל פעם שיוצרים התייחסות לעצם מקדמים את מונה ההתייחסויות של העצם, ובכל פעם שהורסים התייחסות כזאת מפחיתים 1 מהמונה (מאט שימוש בהתייחסויות!)
- כאשר המונה מגיע ל-0, משחררים את הזיכרון של העצם
- בגלל שבג׳אווה מותרים מעגלים בגרף ההתייחסויות, המונה לא תמיד מגיע ל-0 גם אם העצם כבר לא נגיש
- עצם א׳ נגיש משורש ומצביע על ב׳ שמצביע חזרה על א׳
- ביטול ההצבעה מהשורש: שניהם לא נגישים אבל עם מונה 1
- בשימוש במערכות קבצים שבהן אי אפשר ליצור מעגלים

אוספי זבל יותר מתוחכמים

- generational collectors: הזיכרון מחולק לשני אזורים (או יותר), אחד עבור עצמים צעירים והשני עבור ותיקים
- זבל נאסף בתכיפות באזור הצעירים, אבל לעיתים נדירות באזור הותיקים
- עצם צעיר ששורד מספר מסוים של מחזורי איסוף משודרג לאזור הותיקים
- incremental collectors: האוסף לא עוצר את התוכנית לכל זמן האיסוף; האוסף עוצר את התוכנית לזמן קצר, אוסף קצת זבל, והתוכנית ממשיכה לרוץ; מיועד לתוכניות אינטראקטיביות
- concurrent collectors: מיועדים למחשבים מרובי מעבדים; התוכנית ואוסף הזבל רצים במקביל

זיכרון דולף גם אם משתמשים באוסף זבל

- יש עצמים נגישים, כלומר שיש מסלול של התייחסויות משורש אליהם, אבל שהתוכנית לא תיגש אליהם
- אי אפשר לזהות אוטומטית את כל העצמים הללו; זו בעיה לא כריעה, יותר קשה מבעיית העצירה
- דוגמאות נפוצות: מערכים או מבנה נתונים ששומר התייחסויות לעצמים שהתוכנית אכן צריכה, אבל גם לעצמים שהיא לא צריכה יותר; הם לא ישתחררו; צריך השמה ל-null
- התייחסות שאינה null אבל שהפעולה הבאה עליה תהיה השמה
- בשפות שדורשות שחרור מפורש (C ו-C++, למשל) יש עוד סוג של דליפה, של עצמים שאינם נגישים אבל לא שוחררו

גווני אפור

- עד עכשיו השתמשנו בשני צבעים לסימון עצמים: עצמים נגישים (לבנים) ועצמים לא נגישים (שחורים)
- בעצם יש גם עצמים שהם נגישים, אבל אולי אנו מוכנים לוותר עליהם
- בג׳אווה יש שני סוגים כאלה
- למעשה אלו סוגים של התייחסויות שלא גורמות לאוסף הזבל לסמן את העצם כנגיש
- סוג אחד, התייחסויות רכות (soft references), מאפשרות שחרור אם אין התייחסויות רגילות לעצם ואם חסר זיכרון
- סוג שני, התייחסויות חלשות (weak references), גורמות לאוסף הזבל לשחרר את העצם אם אין אליו התייחסויות יותר חזקות (רגילות או רכות)

התייחסויות רכות

```
class CachedFile {
  String url;
  java.lang.ref.SoftReference cache;
  public CachedFile( String url ) {
    this.url = url;
    load();
  }
  private void load() {
    Data content = get it from the given URL
    cache = new SoftReference(content);
  }
```

התייחסויות רכות (המשך)

```
class CachedFile {
  String url;
  java.lang.ref.SoftReference cache;
  ...
  public byte[] get() {
    if (cache.get() == null) load();   reload
    return (Data) cache.get();
  }
}
```

לאוסף הזבל מותר לשחרר עצמים שיש אליהם רק התייחסויות רכות, והוא עושה כך לפני שמודיע על OutOfMemoryError

התייחסויות חלשות

- עצמים שיש אליהם רק התייחסויות חלשות (`java.lang.ref.WeakReference`) מסומן על ידי האוסף כזבל
- שימושי במקרים שבהם רוצים לשמור התייחסות לעצם מבלי שזו תמנע את איסופו; התייחסות בלי בעלות
- דוגמא: המחלקה `WeakHashMap` שמאפשרת לזכור מיפוי, אבל באופן שבו אם אין התייחסויות חזקות או רכות למפתח, המיפוי שקשור למפתח נעלם מאליו והמפתח משתחרר
- מבנה הנתונים הזה מונע דליפת זיכרון בגלל אי-הוצאת המיפוי ממבנה הנתונים
- בדרך כלל עדיף להוציא מפורשות את המיפוי
- לסיכום, סוג התייחסויות לא שימושי כל כך

תורי התייחסויות

- על ידי קשירת התייחסות חלשה/רכה/פאנטום לעצם מסוג `java.lang.ref.ReferenceQueue`, אפשר לקבל מאוסף הזבל מעין הודעה שהעצם המיוחס נאסף
- אם התייחסות רכה/חלשה כזו קשורה לתור, אוסף הזבל מוסיף את ההתייחסות לתור לאחר שהעצם המיוחס נאסף והזיכרון שוחרר; קריאה ל-`get` תחזיר `null`
- התייחסויות מסוג פאנטום (`PhantomReference`) מיועדות אך ורק לקבלת הודעה אודות איסוף של עצם; כמו התייחסויות חלשות הן אינן מונעות איסוף, אבל השירות `get` שלהן תמיד מחזיר `null`

finalize()

- שירות שכל עצם יורש מ-Object
- מופעל על ידי אוסף הזבל לפני שהעצם נמחק סופית
- דריסה שלו מאפשרת לבצע פעולות לפני שחרור; בעיקר שחרור משאבים שהעצם קיבל גישה אליהם (קבצים, למשל)
- עדיף לא להשתמש במנגנון הזה, כי ההפעלה של finalize עלולה להתבצע זמן רב לאחר שהעצם לא נגיש
- סיבוך נוסף נגרם מכך שהפעולה של finalize עלולה להפוך את העצם חזרה לנגיש; במקרה כזה הוא לא ישתחרר
- אם העצם יהפוך ללא נגיש בהמשך, אוסף הזבל ישחרר אותו, אבל לא יפעיל שוב את finalize

טעינה וקישור דינאמי של מחלקות

- תוכנית ג׳אווה יכולה לטעון במהלך הריצה שלה מחלקות באופן סתום או מפורש
- כאשר תוכנית מתייחסת למחלקה חדשה שלא הייתה בשימוש עד לאותו רגע, מתבצעת טעינה אוטומטית של המחלקה; מחלקות אינן נטענות בדרך כלל לפני שיש בהן צורך
- זו טעינה סתומה: התוכנית אינה מבקשת מפורשות לטעון את המחלקה
- היכן ה-JVM מחפש מחלקות? בדרך כלל במדריכים ומארזי jar שמוגדרים בתור ה-class path של התוכנית; ניתן לקבוע אותו על ידי משתנה סביבה או על ידי ארגומנט ל-JVM
- אבל לפעמים ה-JVM מחפש במקומות אחרים, והתוכנית גם יכולה לטעון מחלקות באופן יזום ובאופן מפורש

טוענ‌י מחלקות (class loaders)

- טעינה של מחלקות מתבצעת בעזרת הרחבות של המחלקה המופשטת `java.lang.ClassLoader`
- המחלקה המופשטת הזו מגדירה שירותים שמקבלים ייצוג של מחלקה בפורמט של קובץ בינארי (קובץ class.) ובונים ייצוג שלה בתוך ה-JVM, ייצוג שבעצמו מיוצג על ידי עצם מהמחלקה `java.lang.Class`
- שני השירותים הללו הם `defineClass` ו-`resolveClass`
- מחלקות מרחיבות מגדירות את השירות `loadClass` שתפקידו למצוא את הייצוג הבינארי ולטעון אותו בעזרת שני השירותים של המחלקה המופשטת

היררכיה של טועני מחלקות

- כאשר ה-JVM מתחיל לפעול, הוא משתמש בטוען סטנדרטי שהשירות loadClass שלו מחפש קבצי class במדריכים ומארזי ה-jar שמוגדרים ב-class path
- הטוען הסטנדרטי הזה מוחזר על ידי `ClassLoader.getSystemClassLoader`
- בהמשך התוכנית עצמה יכולה להגדיר טוענים נוספים; לכל טוען יש הורה; אם לא מצהירים על הורה מפורש ההורה הוא הטוען הסטנדרטי
- הטוענים הנוספים יכולים לטעון מחלקות מהרשת, ממארזי jar או מארזים אחרים שהתוכנית מגלה שהיא צריכה בזמן ריצה, או אפילו יכולים לייצר bytecode בעצמם

טעינה סתומה והיררכיית הטוענים

- נניח שמחלקה נטענת על ידי טוען מיוחד מקובץ jar
- בהמשך המחלקה הזו מתייחסת למחלקות נוספות, וצריך לטעון אותן באופן סתום; מהיכן הן ייטענו?
- בדרך כלל, חלק מהמחלקות הנוספות הן מחלקות סטנדרטיות של ג׳אווה או של התוכנית, וצריך לטעון אותן באופן רגיל, אבל חלק יימצאו בעצמן בתוך אותו jar
- הפתרון: כל מחלקה זוכרת את הטוען שטען אותה, והוא זה שטוען מחלקות שהיא מתייחסת אליהן, גם בטעינה סתומה
- הטוען המיוחד ינסה קודם כל לטעון בעזרת ההורה שלו, ואם ההורה נכשל, הוא ינסה לטעון את המחלקה הדרושה בעצמו
- זה מממש את ההתנהגות הרצויה

למה טעינה מפורשת?

- טעינה מפורשת של מחלקה מאפשרת ליצור עצם מהמחלקה Class שמייצג מחלקה עם שם נתון (כמחרוזת)
- זה שימושי כאשר קובץ בקרת תצורה מורה לתוכנית לטעון תוסף (plugin) ולהפעיל מחלקה מסוימת
- נדון בכך בהמשך הקורס, אבל כעת נאמר שזה מאפשר מידה גדולה של מודולאריות, כולל אפשרות למפתחי תוכנה להוסיף יכולות לתוכנה קיימת שלא הם כתבו ושלא הם ארזו

- טעינה דינאמית, גם סתומה, מקצרת את זמן האתחול של תוכנית ומאפשרת אריזה גמישה במספר מארזים
- גם כאן אפשר לחשוב על אופטימיזציות (למשל טעינה חמדנית של מחלקות נפוצות)

סיכום נושא מנגנוני השפה

- תוכנה מופצת כ-bytecode; מורצת על ידי פרשן או מתקמפלת בזמן ריצה לשפת מכונה "ילידה"
- הפעלה של שגרה דרך מנשק היא מסובכת; לא צריכה להיות איטית, אבל בפועל לעיתים איטית
- יש הבדלי ביצועים גדולים בין JVM-ים שונים (JIT, מנשקים)
- איסוף זבל אוטומטי מפחית את כמות הפגמים בתוכנית, אבל יש לו מחיר בזמן ריצה ו/או בזיכרון; המחיר גדול במיוחד כאשר מספר גדול של עצמים קטנים חיים לאורך זמן
- תוכנית יכולה להשפיע על אוסף הזבל על ידי קריאה לו, על ידי שימוש בסוגי התייחסויות שלא מונעות איסוף, וע"י finalize
- טעינה דינאמית של מחלקות מקנה גמישות ומאפשרת ליצור תוספים לתוכניות (התוכניות צריכות לתמוך בכך)