נושאים מתקדמים בתכנות מונחה עצמים

הרצאה 8

פרופ' עפר שיר

ofersh@telhai.ac.il

החוג למדעי המחשב

תל-חי

המכללה האקדמית

מבנה ההרצאה

- Introduction to Design Patterns
- OOD Concepts
- Notation
  - UML and its history
- The MVC model
- Recap and Outlook

Sources:
Gamma et al., “Design Patterns – Elements of Reusable Object-Oriented Software” (1995)
D. Kieras, EECS U-Michigan

Introduction

Design Patterns

מוטיבציה

- תכנון תוכנה מונחית-עצמים זו משימה קשה.
- תכנון תוכנה מונחית-עצמים, בה ניתן לעשות שימוש חוזר (*reusable software*), זו משימה קשה יותר.
- מתכננים מנוסים מבצעים תכנון טוב.
- אנשי תוכנה חדשים נוטים לרגרסיה לטכניקות שאינן מונחות-עצמים.
- איזה ידע וכלים יש למתכננים המנוסים?

מוטיבציה

- מתכננים מנוסים יודעים לא לפתור כל בעיה מהיסוד: הם עושים *שימוש חוזר בפתרונות קיימים*.
- מתוך דברים של כריסטופר אלכסנדר:

"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"

- מי זה אלכסנדר וכיצד הוא קשור?!?
  - אנלוגיה I.
- אנלוגיה II: טלנובלות!

מהו Design Pattern ?

- פרדיגמה (תבנית) לפתרון בעיית תוכנה אשר צצה לעיתים קרובות
  - איננה מהווה פתרון סגור או קוד מוכן
- קיימות 23 תבניות קאנוניות שהוכחו כנכונות ויעילות במקרים רבים
- יתרונות רבים, בעיקר זירוז תהליך הפיתוח ובהירותו:
  - פתרונות שלא פותחו עד סופם עלולים להיתקל בקשיים שאינם ניכרים מראש, אותם DP מוכחים מונעים
  - הקוד הופך קריא יותר אם הכותבים והקוראים שולטים ב-DP

Conceptual

OBJECT-ORIENTED DESIGN

Object’s Class vs. Type

<u>יש להבחין בין המחלקה של האובייקט לבין טיפוסו:</u>

- מחלקה של אובייקט מגדירה את מימושו
  - מצבו הפנימי ומימוש הפעולות שלו
- טיפוס של אובייקט מגדיר את המנשק שלו
  - קבוצת הבקשות (*requests*) אליהן הוא יכול להגיב
- אובייקט יכול להיות שייך למספר טיפוסים, ואובייקטים של מחלקות שונות יכולים להיות מאותו טיפוס.
- שפות כמו C++ עושות שימוש במחלקות בכדי להגדיר גם את טיפוס האובייקט וגם את מימושו.

Class vs. Interface Inheritance

<u>יש להבחין בין הורשת מחלקה לבין הורשת מנשק:</u>

- הורשת מחלקה מגדירה מימוש של אובייקט באמצעות מימוש של אובייקט אחר
  - מנגנון לשיתוף של קוד וייצוג
- הורשת ממשק (או *subtyping*) מאפיינת מצב בו אובייקט ניתן לשימוש במקום אובייקט אחר
- קל לבלבל בין המושגים, בפרט מכיוון ששפות כמו C++ אינן עושות את ההבדלה באופן מפורש
  - בפועל, אין הבדל בין שני המושגים בהורשת C++

Programming to an Interface

- הורשת מחלקה היא מנגנון שימושי להרחבת פונקציונליות; מקבלים "מימוש נוסף כמעט בחינם"
- אבל שימוש חוזר בפונקציונליות הוא רק מחצית מן הסיפור – היכולת של הורשה להגדיר משפחות של אובייקטים עם מנשקים זהים היא גם-כן חשובה
  - מדוע? מפני שפולימורפיזם תלוי בזה!
- שימוש זהיר (ונכון) בהורשה יביא למצב בו כל המחלקות היורשות ממחלקה אבסטרקטית חולקות את המנשק שלה
  - במקרה זה, כל הצאצאים יגיבו לבקשות המופיעות במנשק של המחלקה האבסטרקטית – הם כולם subtypes

Programming to an Interface (cont’d)

קיימים שני יתרונות לפעולות על אובייקטים המתבצעות רק באמצעות המנשקים המוגדרים במחלקות אבסטרקטיות:

- נותרת עמימות אצל הקלאיינטים באשר לטיפוסי האובייקטים בהם הם עושים שימוש, כל עוד האובייקטים תואמים למנשק שהקלאיינט מצפה
- הקלאיינטים *נותרים בחשיכה* גם באשר למחלקות הממשות אובייקטים אלו. הקלאיינט מודע רק למחלקות האבסטרקטיות המגדירות את הממשק.

Reusable Object-Oriented Design

- היתרונות הללו מפחיתים משמעותית את תלויות המימושים של תתי-מערכות – כך שהדבר הוביל לניסוח העיקרון הבא בתכנון מונחה-עצמים:

Program to an interface, not an implementation.

🡸 Creational patterns יבטיחו שהמערכת תיכתב באמצעות *מנשקים* ולא *מימושים*.

Inheritance vs. Composition

שתי הטכניקות הנפוצות לשימוש חוזר בפונקציונליות במערכות מונחות-עצמים הן הבאות:

(1) הורשת מחלקות (או subclassing)

(2) הרכבת אובייקט (object composition)

- שימוש חוזר באמצעות (1) נחשב ל-*white-box reuse* בשל הנראות של המחלקות הנורשות
- שימוש חוזר באמצעות (2) נקרא *black-box reuse* – בשל האנקפסולציה של האובייקט המוכל
- קיימים יתרונות וחסרונות בהיבטים שונים עבור שתי הטכניקות הללו

Class Inheritance

- הורשת מחלקה מהווה מרכיב סטטי שנסגר בשלב הקומפילציה; השימוש במנגנון הוא קל ומיידי, כולל האפשרות לעדכון הפונקציונליות באמצעות דריסה.
- בהתאם, לא ניתן לשנות מימושים בזמן-ריצה (חיסרון: חוסר גמישות)
- תלות במימוש ובייצוג של מחלקת הבסיס – שבירת עיקרון האנקפסולציה:
  - שינויים אצל ההורה עשויים לגרור שינויים אצל הצאצאים
  - התלות במימוש עלולה לגרום לאי-התאמה לבעיות חדשות
  - פתרון אפשרי: ירושה רק ממחלקות אבסטרקטיות

Object Composition

- הרכבת אובייקט מהווה רכיב דינאמי שמתרחש בזמן-ריצה (קבלת רפרנסים/מצביעים לאובייקטים אחרים)
- הדרישה המעשית היא לכבד מנשקים קיימים
  - תכנון המנשקים מהווה אתגר לא קל
  - מנגד, עיקרון האנקפסולציה אינו נשבר
  - כל אובייקט יכול להיות מוחלף באמצעות אובייקט אחר, כל עוד הם מאותו **טיפוס**.
  - קיימות פחות תלויות במימושים
- ההיראכיות נותרות קטנות, אובייקטים נשארים בקפסולה ובפוקוס – לא מקבלים "מפלצות הורשה"

2nd principle of object-oriented design

הדברים האמורים מובילים לעיקרון הבא:

Favor object composition over class inheritance.

הטענה הכללית היא שהשימוש בהורשה הינו רב מדי, פשוט מפני שהפך לטכניקה מוכרת ופופולארית, אך לאו דווקא מפני שהוא יותר מותאם לשימושים חוזרים.

אנחנו צפויים לראות שימוש נרחב בהרכבת אובייקטים ב-DP.

NOTATION

Unified **M**odeling **L**anguage

- שפת מידול כללית המשמשת מפתחי תוכנה
- המטרה העיקרית: ויזואליזציה של תכנון המערכת
- מטרות נוספות: סטנדרטיזציה של תהליכי תכנון (!)
- פותחה ע"י Booch-Jacobson-Rumbaugh באמצע שנות ה-90 של המאה ה-20.
- אומצה ע"י **O**bject **M**anagement **G**roup ב-1997.
- פורסמה ע"י International Organization for Standardization כסטנדרט ISO בשנת 2005.

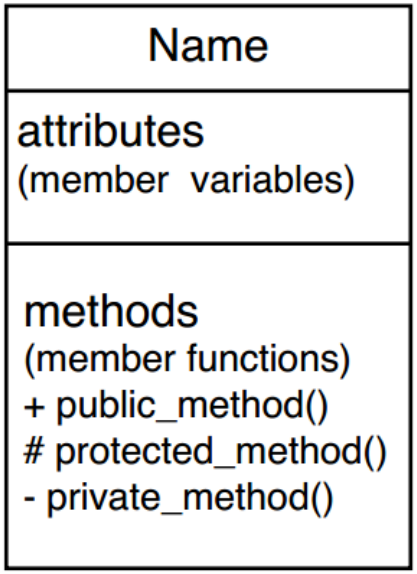
Rational Software

- Booch-Jacobson-Rumbaugh פיתחו את UML במסגרת עבודתם ב-Rational
- המוצר המקורי היה סביבת פיתוח עבור שפת Ada
- רכישה של חברת תוכנה אחרת, בעלת מוצר בשם Rhapsody למידול *Statecharts*, הביאה לבסוף לרכישת החברה ע"י IBM.
- Rational Rhapsody היא סביבת מידול ויזואלית, מבוססת UML, המיועדת לתכנון מורכב ומרובה-תכננים, ומסוגלת ליצור קוד בשפות C/C++/Java/C#

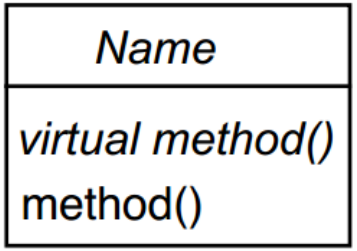
IBM® Rational® software

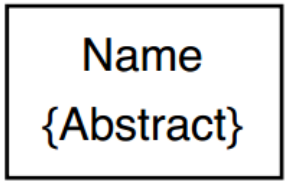
UML in a Nutshell (1)

Class

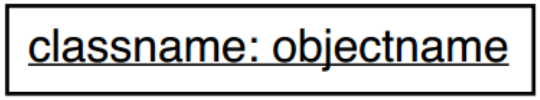


Abstract class

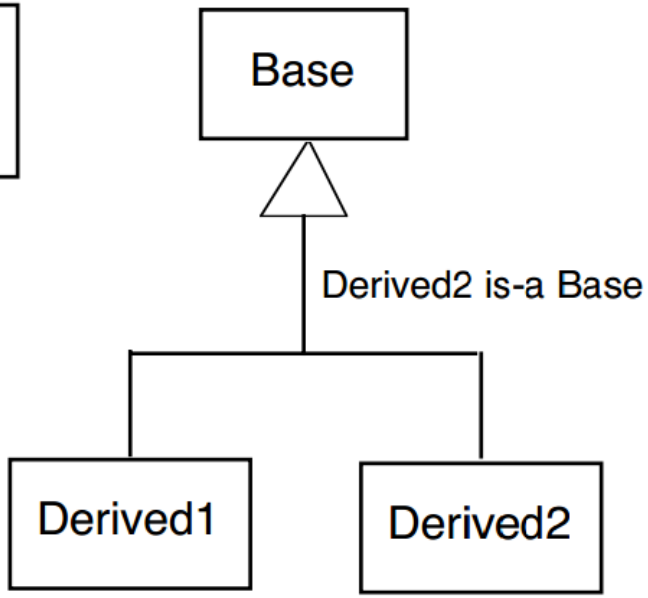




Object

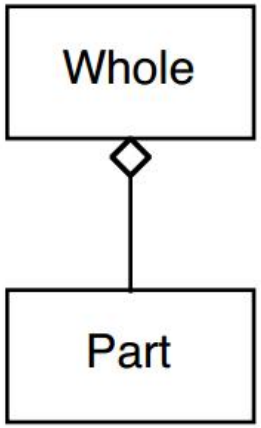


Inheritance (is-a) relationship

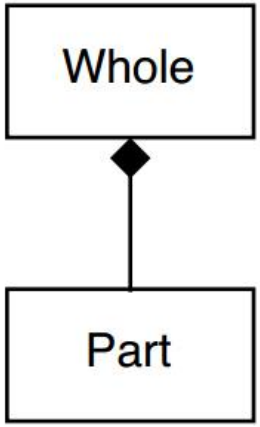


UML in a Nutshell (2)

Aggregation and Composition (has-a) relationship

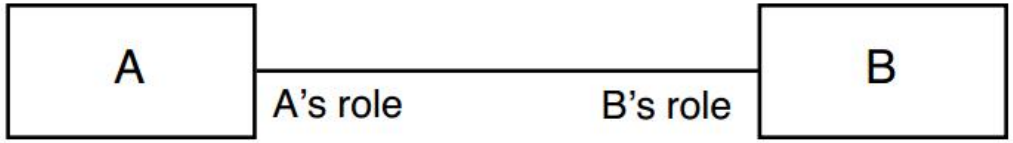


Whole has Part as a part;
lifetimes might be different;
Part might be shared with other Wholes.
(aggregation)

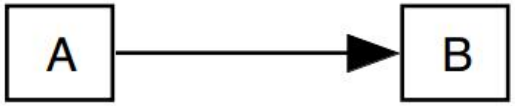


Whole has Part as a part;
lifetime of Part controlled by Whole,
Part objects are contained in one Whole object.
(composition)

Association (uses, interacts-with) relationship

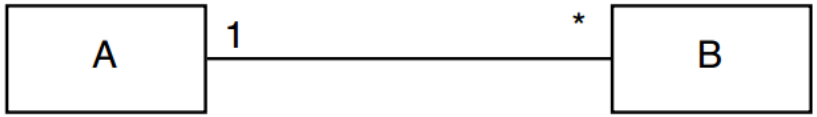


Navigability - can reach B starting from A



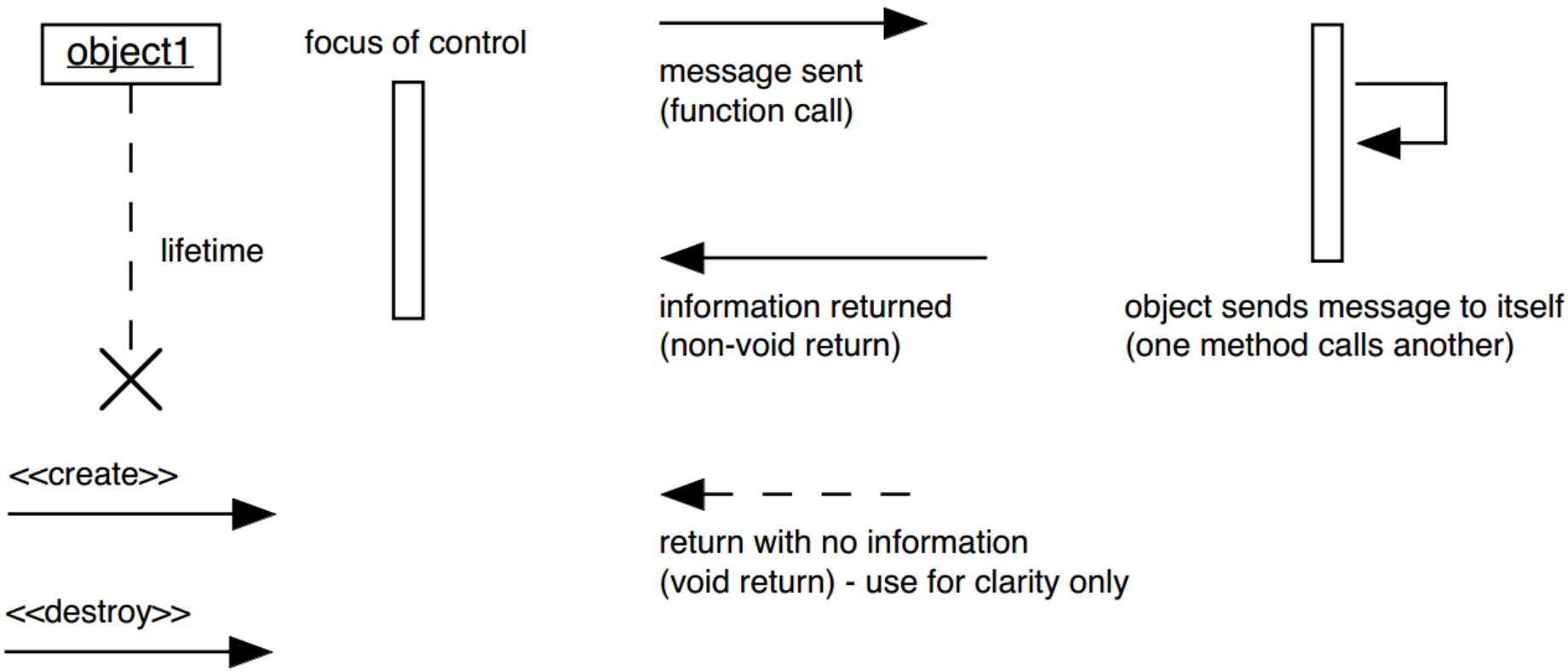
Multiplicity in Aggregation, Composition, or Association

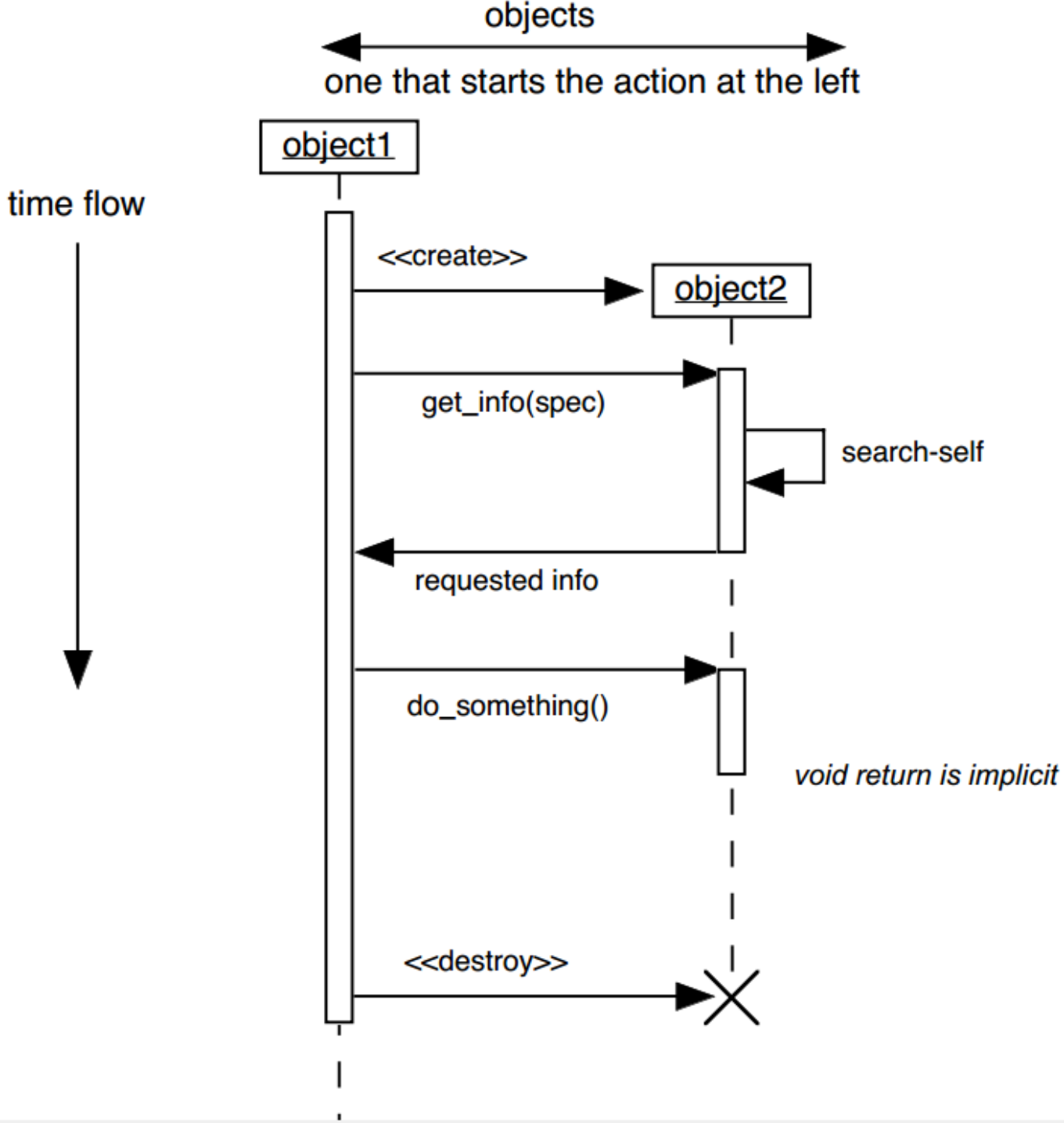
* - any number	0..1 - zero or one	Follow line from start class to end class,
1 - exactly 1	1..* - 1 or more	note the multiplicity at the end.
n - exactly *n*	*n* .. *m* - *n* through *m*	Say "Each <start> is associated with <multiplicity> <ends>"

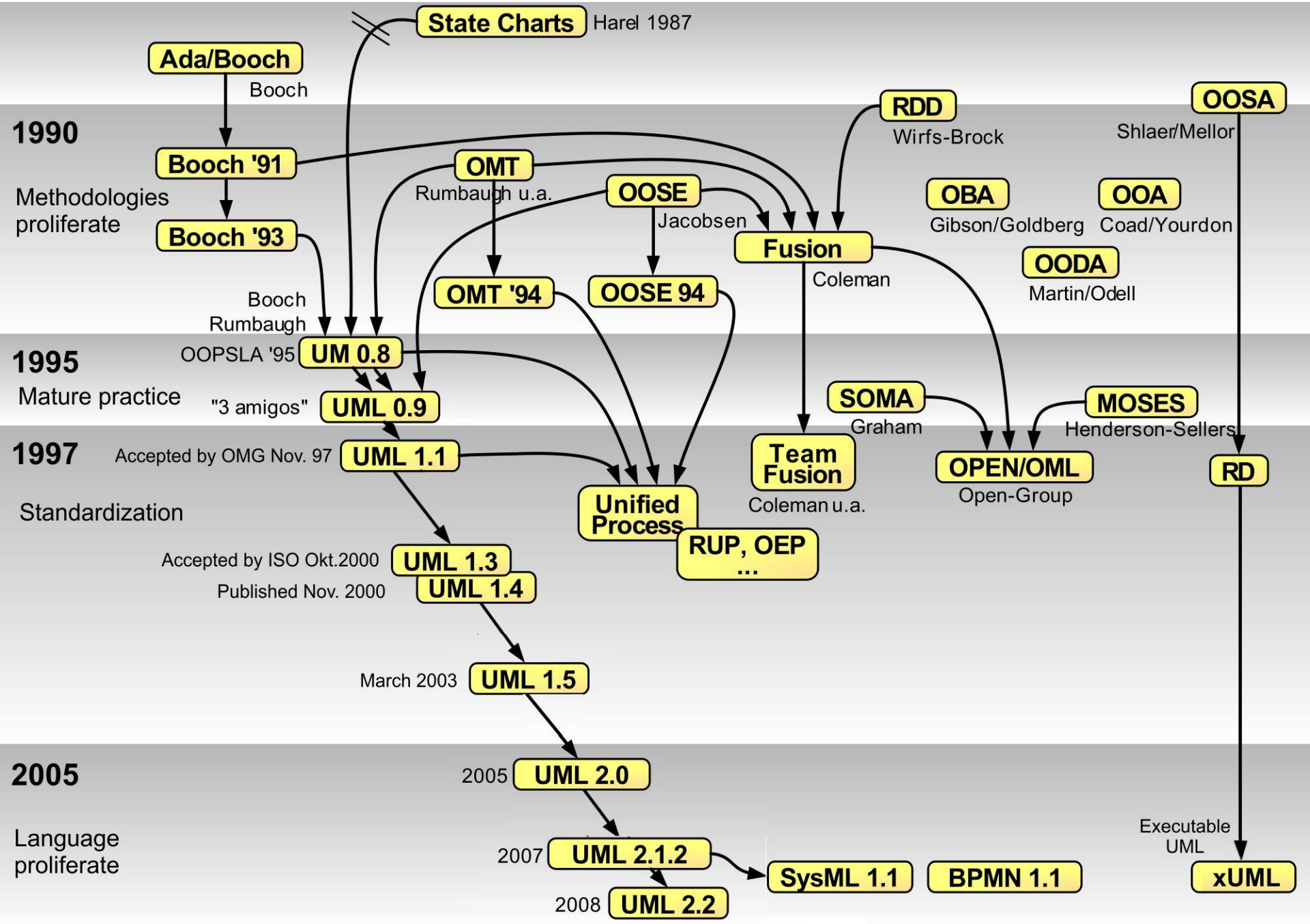


Each A is associated with any number of B's.
Each B is associated with exactly one A.

UML in a Nutshell (3)






State Charts
Harel 1987
Ada/Booch
Booch
1990
Booch '91
Methodologies
proliferate
Booch '93
OMT
Rumbaugh u.a.
OOSE
Jacobsen
RDD
Wirfs-Brock
OOSA
Shlaer/Mellor
OBA
Gibson/Goldberg
OOA
Coad/Yourdon
Fusion
Coleman
OODA
Martin/Odell
OMT '94
OOSE 94
Booch
Rumbaugh
OOPSLA '95
UM 0.8
1995
Mature practice
"3 amigos"
UML 0.9
SOMA
Graham
MOSES
Henderson-Sellers
1997
Accepted by OMG Nov. 97
UML 1.1
Team
Fusion
Coleman u.a.
OPEN/OML
Open-Group
RD
Standardization
Unified
Process
RUP, OEP
...
Accepted by ISO Okt.2000
UML 1.3
Published Nov. 2000
UML 1.4
March 2003
UML 1.5
2005
2005
UML 2.0
Language
proliferate
2007
UML 2.1.2
SysML 1.1
BPMN 1.1
Executable
UML
xUML
2008
UML 2.2

THE MVC MODEL

The **M**odel-**V**iew-**C**ontroller Problem

- קיימים נתונים גולמיים בשיתוף גורמים שונים
- נדרשת תמיכה במספר הצגות (*views*)
  - *HTML*, *JFC/swing*, *JSON*, *XML* –
- עדכונים מתרחשים דרך אינטראקציות שונות
- העבודה השוטפת (קריאה/עדכון) אינה אמורה להשפיע על התנהגות האפליקציה הראשית

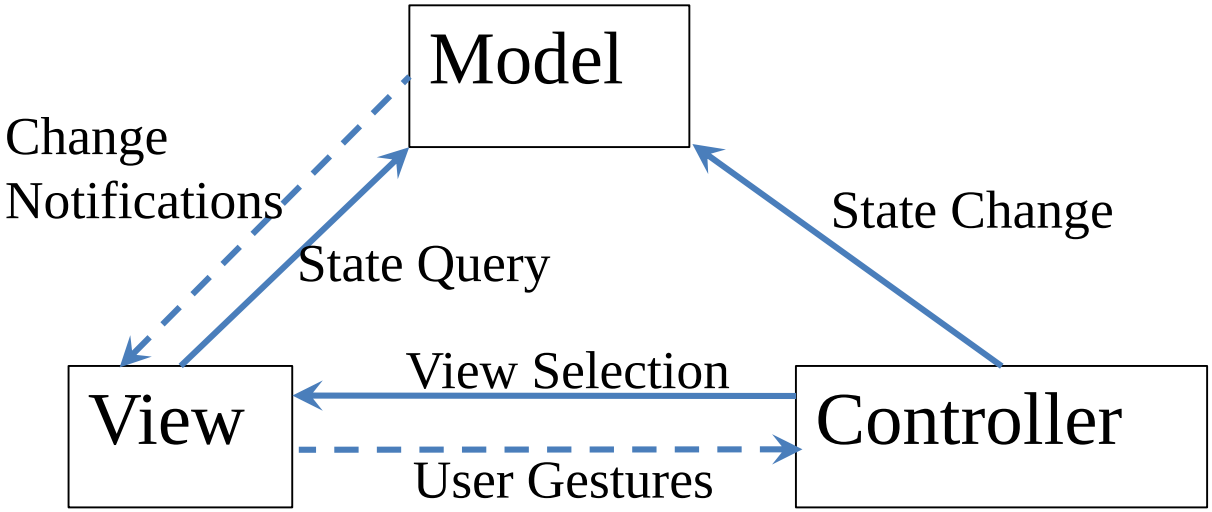
MVC: Solution

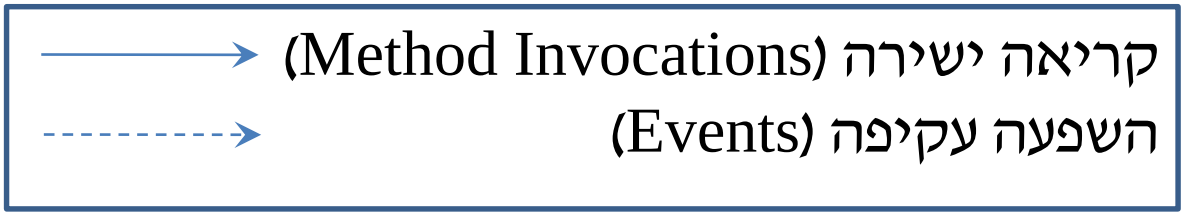
- הפרדה של פונקציונליות המודל מן התצוגה (*presentation*) ומהלוגיקה של הבקרה (*control*).
- באופן זה, מספר הצגות שונות (*views*) יכולות לחלוק את אותו מודל נתונים.
- פתרון זה מאפשר מימוש, בדיקה ותחזוקה קלים יותר עבור מערכות התומכות במספר קלאיינטים.

MVC: Responsibilities

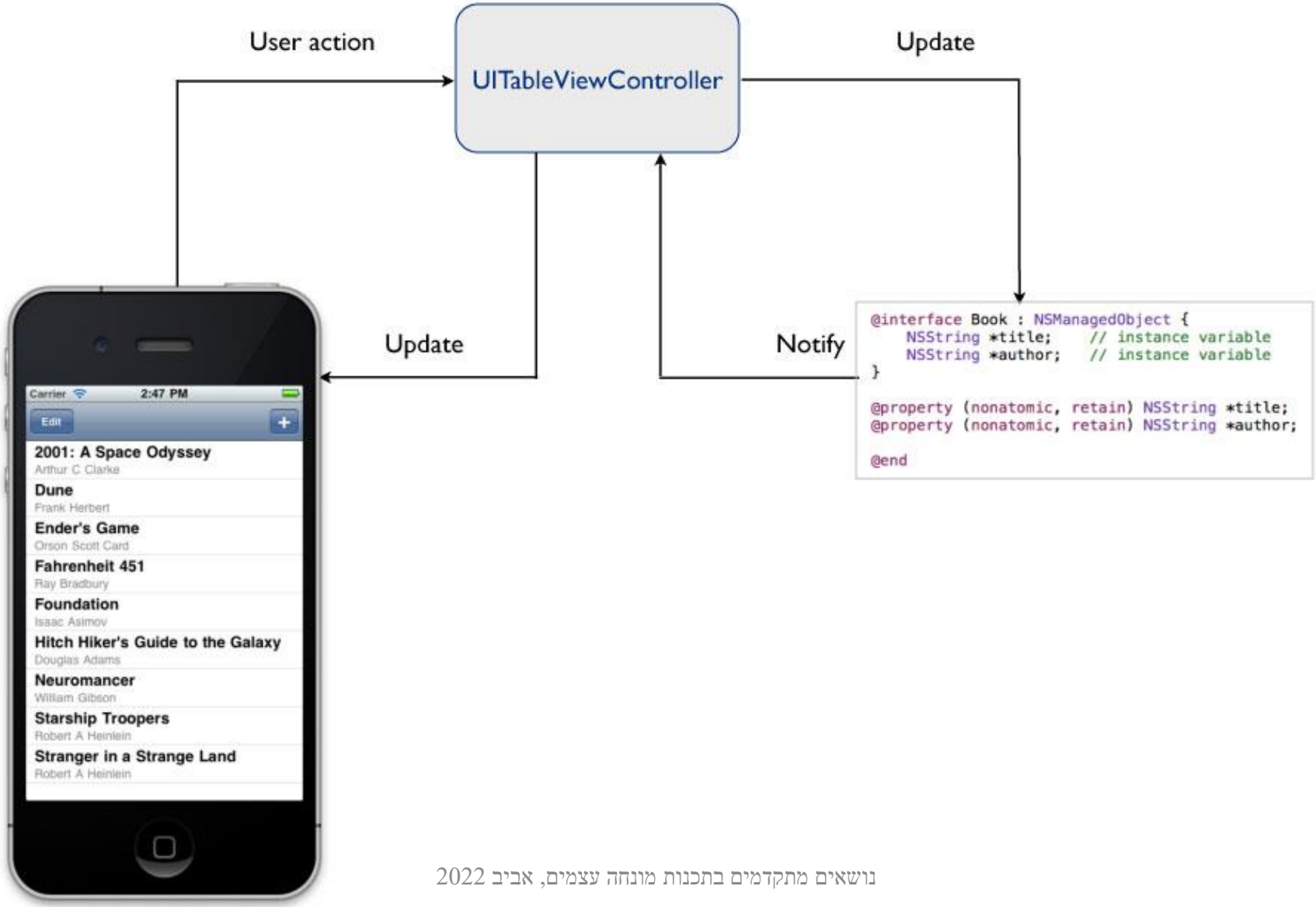
- ***Model***
  - אחראי על ייצוג הנתונים ברמה הגולמית
  - בעל הרשאת שליטה על גישה ועדכון הנתונים
- ***View***
  - אחראי על ביטוי תכני המודל
  - בעל הרשאת גישה לנתונים **דרך *model***
  - מכתיב את פרטי תצוגת הנתונים
- ***Controller***
  - אחראי על הגדרת תגובת האובייקט לקלט משתמש
  - מתרגם אינטראקציות *view* לפעולות *model*
  - עשוי לשמש כמתרגם גם בכיוון השני

MVC


Model
Change
Notifications
State Query
State Change
View Selection
View
Controller
User Gestures


קריאה ישירה (Method Invocations)
השפעה עקיפה (Events)

MVC used in iOS

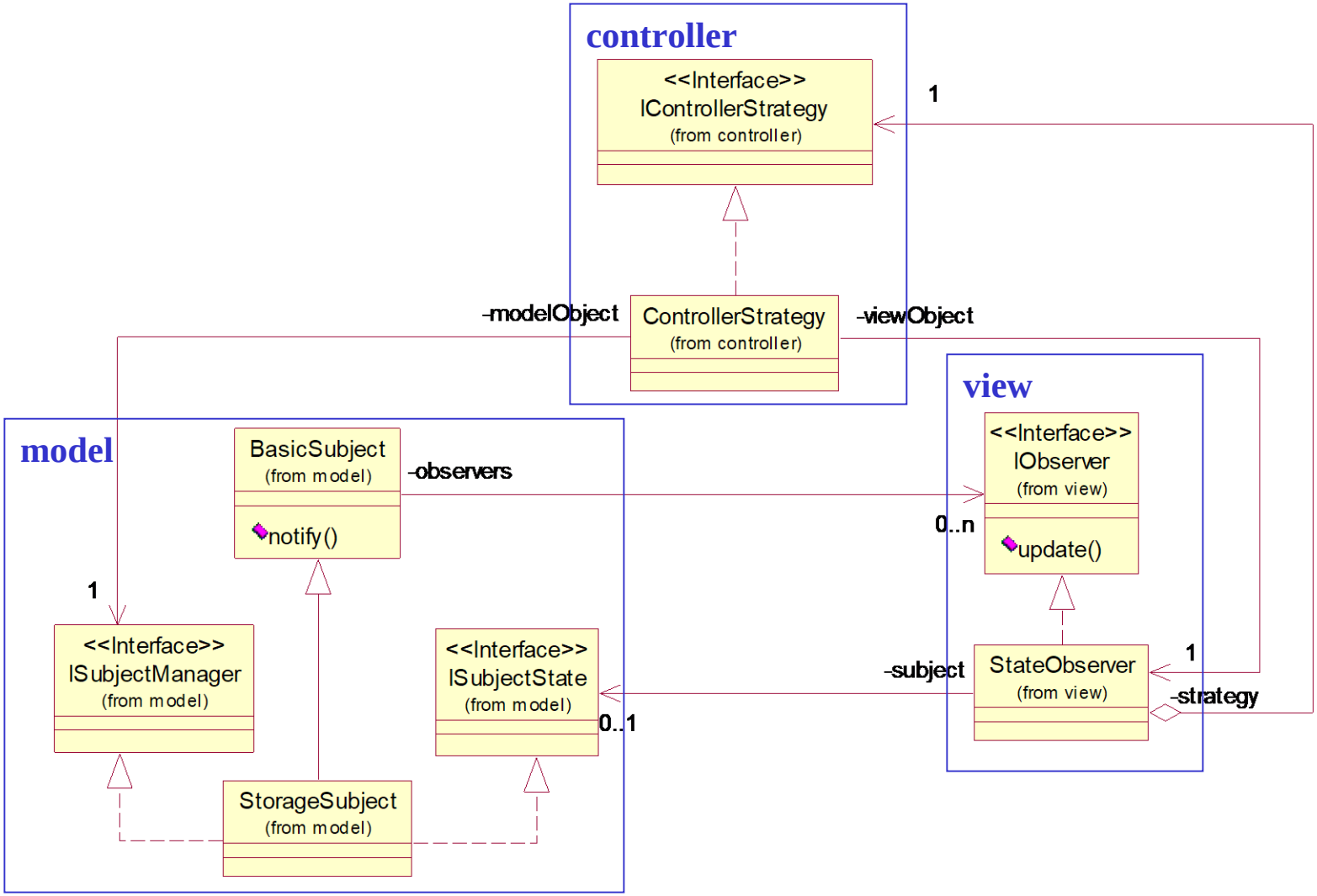


MVC *featuring* DP

MVC אינו נחשב ל-DP קאנוני – אלא למודל מאקרו כללי, אשר יכול לשלב מספר DP קאנוניים:

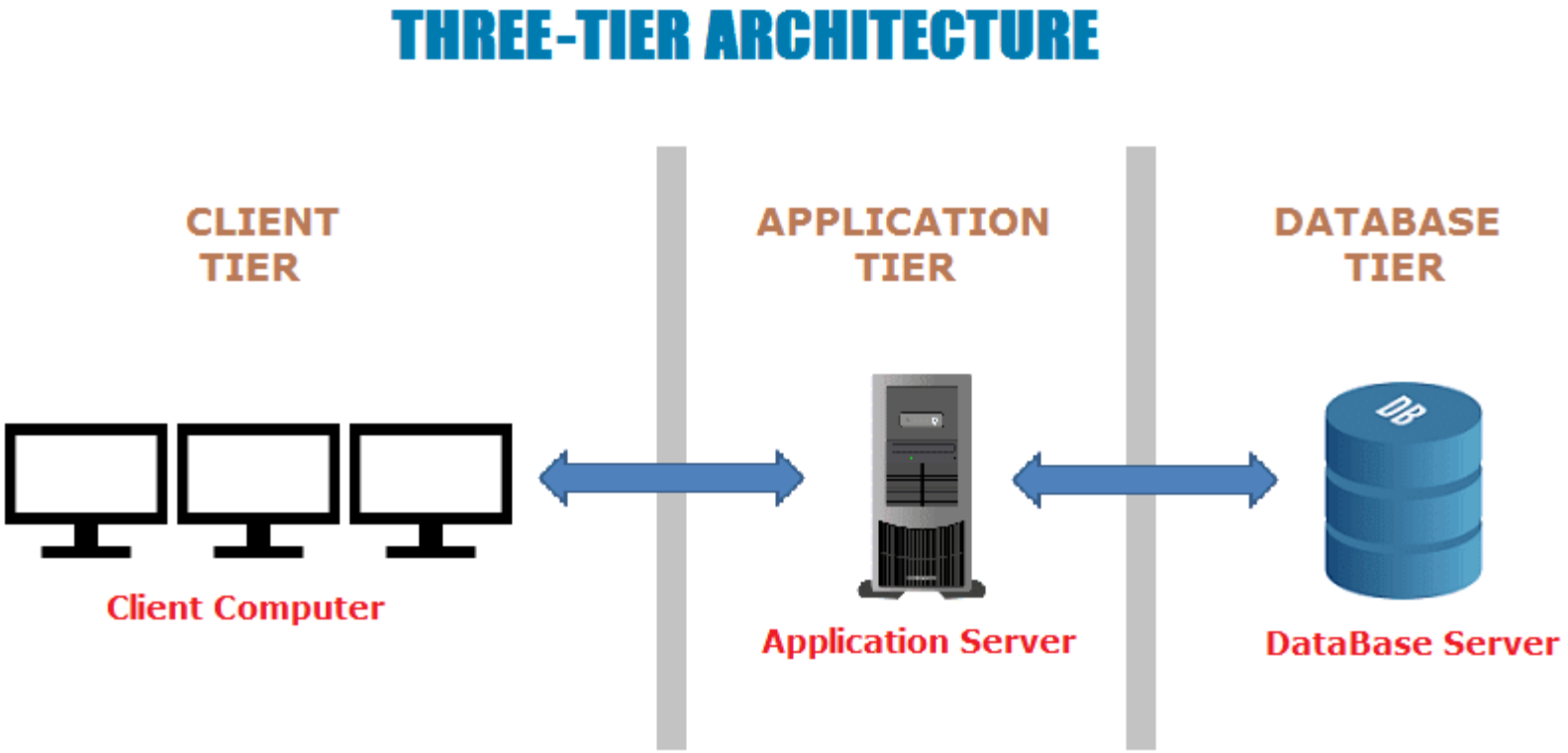
- עניין הפרדת ההצגה מן המודל בא לידי ביטוי בתבנית Observer
- שילובים של תצוגות ובקרות שונות אפשריים בעזרת תבנית Composite
- יחסי View-Controller יכולים להיות מתוארים באמצעות תבנית Strategy
- Factory Method, Decorator וייתכנו אף נוספים...

MVC Class Diagram



MVC Variants

ארכיטקטורת שלוש השכבות:



RECAP & OUTLOOK

Patterns vs. Idioms

- Design patterns הם תבניות תכנון ע"פ עקרונות מופשטים, והם אינם תלויי שפה; *language agnostic*
  - Singleton
  - Factory Method
- Idioms ("צירופי מילים") הם ביטויים או מרכיבים תלויי-שפה, אשר בדרך כלל מתייחסים לתבניות בסדר גודל קטן
  - "Virtual Constructor"
  - "Wrapper"
  - "Compiler Firewall"

Scope and Generalization

- DP אינם תלויי שפה
- אך כאלה שמסתמכים על תמיכה OOP:
  - הורשה
  - מנשק
  - פולימורפיזם
- יתכנו DP שאינם ישימים או אינם יעילים בשפות מסוימות!
- נעסוק בעיקר בשפת C++, אך לעיתים נראה Java

Design Pattern Space by G.o.F.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class** (inherit./ static)	• Factory Method	• Adapter	• Interpreter • Template Method
	Object (dynamic)	• Abstract Factory • Builder • Prototype • Singleton	• Adapter • Bridge • Composite • Decorator • Facade • Proxy	• Chain of Responsibility • Command • Iterator • Mediator • Memento • Flyweight • Observer • State • Strategy • Visitor

עקרונות מפתח לשימוש ב-DP

- יש לקחת את יחסי המחלקות והפרטים הקטנים ברצינות רבה!
- שימוש בשם מחלקה המזכיר DP קאנוני לא ישיג את המטרה – רק שימוש מדויק באותו מבנה / היראַרכיה יביא איתו את הפתרון החזק של DP
- יש לזכור כי מטרת מרבית ה-DP היא הכשרת הקרקע להרחבות / הכללות עתידיות – במחיר של קוד רב יותר ואולי תקורה בביצועים
- מטרת השימוש היא לא קוד קומפקטי או מהיר, אלא קוד קל להרחבה/הכללה

– אין לשבור את התבנית לשם השגת יעילות – זה מפספס את כל הרעיון!

Outlook

- Creational (16/05)
- Structural (23/05)
- Behavioral (30/05)