גירסה 1.00 – 10.3.2005

תכנות מונחה עצמים

ניר אדר

מסמך זה הורד מהאתר http://underwar.livedns.co.il.

Nir Adar

Email: underwar@hotmail.com

Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

הקדמה

מסמך זה עוסק בנושא **תכנות מונחה עצמים**. תכנות מונחה עצמים היא טכניקה של הנדסת תוכנה, המספקת מספר כלי הפשטה למתכנת, שהעיקריים בהם הם אובייקטים ומחלקות.

מטרת מסמך זה היא להכיר לקורא את מנגנוני ההפשטה השונים אותם תכנות מונחה עצמים מספק לנו. המסמך מניח ידע בסיסי בעקרונות התכנות מונחה העצמים, למרות שעקרונות תכנות מונחה העצמים מוסברים במסמך זה. כמו כן, המסמך מניח ידע מוקדם נרחב בשפת C++, וכן מביא מספר דוגמאות הנוגעות לשפת Little Smalltalk, אם כי ניתן להבין את המסמך גם ללא הכרות מעמיקה עם שפה זו.

אני מניח שרבים מקוראי המסמך הם סטודנטים בטכניון הלוקחים את הקורס "תכנות מונחה עצמים", ולכן אספק מספר הסברים על הקשר בין מסמך זה לחומר הקורס.
המקורות למסמך זה הינם: סיכומי הרצאות של ד"ר יוסי גיל, סיכומי הרצאות של ד"ר יחיאל קימחי, חומרים רבים מרשת האינטרנט וכן חומרים שכתבתי בעצמי. לא בכל מקום הסכמתי עם מה שנאמר על ידי המרצים בכיתה, ולפיכך מסמך זה משקף את הדעה שלי לגבי תכנות מונחה עצמים.
מסמך זה אינו כולל את כל החומר שהועבר בקורס הנ"ל, אלא את הנושאים שנראו לי רלוונטיים וחשובים. אי לכך, אני בהחלט מקווה שמסמך זה יעזור לכם במהלך הלימודים בקורס, אולם הוא איננו חומר עזר עוקב לקורס. כמו כן, מפאת מגבלות זמן, המסמך אינו כולל את נושאי ההורשה המרובה ואת נושא הגנריות.

ניר אדר,
מרץ 2005

תוכן עניינים

1. מבוא

1.1. מהו תכנות מונחה עצמים

תכנות מונחה עצמים היא טכניקה של הנדסת תוכנה, המספקת מספר כלי הפשטה:

- עצמים (רמת ההפשטה הנמוכה ביותר).
- מחלקות (רמת ההפשטה החשובה ביותר – מחלקה היא הפשטה של אובייקט)
- הורשה – שהיא הפשטה מעל מחלקות.
- גנריות – סוג נוסף של הפשטה מעל מחלקות (ב-C++ הפשטה זו מתבטאת בתבניות).

מטרת מסמך זה היא להכיר לקורא את מנגנוני ההפשטה השונים אותם תכנות מונחה עצמים מספק לנו. המסמך מניח ידע בסיסי בעקרונות התכנות מונחה העצמים, למרות שהוא חוזר ומפרט אותם שוב. במסמך נתרכז בכל אחד כלי ההפשטה שצויינו, נסביר אותו לפרטיו ואת האספקטים השונים בו.

היעדים שתכנות מונחה עצמים מציב לעצמו:

- תוכנה נקייה מבאגים.
- תוכנה הניתנת לשינויים.
- Reusability – ניתן להשתמש בחלקים מהתוכנה שוב בתוכניות חדשות.
- Portability – התוכנה יכולה לרוץ בסביבות שונות.
- יעילות – ביצועים טובים.

כאשר המטרה העיקרית הינה יצירת תוכנה איכותית. תכנות מונחה עצמים מנסה לתת שיטה כללית לפתור בעיות בתהליך יצירת התוכנית כדי שהתיכון, הכתיבה והתחזוקה של התוכנית יהיו קלים.

התכנות לפי עקרונות התכנות מונחה העצמים נותן יתרונות רבים למתכנת. ניתן למנות ביניהן למשל יצירת רכיבים לשימוש חוזר, הסתרת אינפורמציה, encapsulation, הפיכת התוכנית למודולרית יותר ועוד. עם זאת, יתרונות אלו ניתנו גם על ידי גישות שקדמו לתכנות מונחה עצמים.
היכולת המרכזית שגישת התכנות מונחה העצמים מוסיפה לנו מעבר למתודולוגיות האחרות היא **הורשה**.

1.2. פרדיגמות תיכנות

פרדיגמה (paradigm) אלו סוגי ההפשטות בהם המתכנת משתמש.

ה-paradigms העיקריות (לפי טכניקות התיכנות השונות):

תכנות פרוצדורלי:	אלגוריתמים.
תכנות מונחה עצמים:	מחלקות ואובייקטים.
תכנות לוגי:	מטרות (חישובי פרדיקטים).

רוב המתכנתים עובדים רק עם שפה אחת ומשתמשים רק בסגנון אחד. לפיכך הם משתמשים רק בצורת מחשבה אחת ולא יודעים להעריך פתרון בעזרת טכניקה אחת או אחרת. על מנת להיות מתכנתים טובים נרצה להכיר כמה שיותר צורות חשיבה שונות, כדי להתאימן לבעיות שונות. גם אם נכיר פרדיגמה מסוימת ולא נשתמש בה, לעתים נוכל לשלב רעיונות שקשורים אליה בתוכניות שלנו, ולכן תמיד כדי להכיר את צורות החשיבה השונות האפשריות לפתרון בעיה.

התכנות הפרוצדורלי שם דגש על אלגוריתמים ומבני נתונים. בעבר אנשים טענו כי נושאים אלו הינם מהות התוכניות שאנו כותבים. עם הזמן, אנשים מגיעים למסקנה כי ברוב התוכניות הקיימות זהו איננו המצב. ברוב התוכניות – רוב זמן ריצת התוכנית של התוכנית מתרכז בהעברת נתונים ממקום למקום, ופחות בביצוע מניפולציות עליהם (אלגוריתמים). מכאן שהדגש בתוכנית צריך להיות יותר על הנתונים עצמם. תכנות מונחה עצמים מנסה לתת את הדגש הנ"ל.

1.3. אלמנטים של תכנות מונחה עצמים

אלמנטי מפתח של תכנות מונחה עצמים:

- אובייקטים – מאחדים נתונים וקוד ביחד.
- מחלקות – מציגות הפשטה מעל האובייקטים.
- הורשה – מציגה הפשטה מעל המחלקות.

אלמנטים נוספים:

- Dynamic Binding.
- Genericity – הפשטה נוספת מעל מחלקות.

1.4. סיבות לשימוש בתכנות מונחה עצמים

לפי סטטיסטיקות אחוזים עצומים מפרויקטים של תוכנה אינם מגיעים לסיום. גם פרויקטים שהסתיימו, במקרים רבים לא משתמשים בהם או שמשתמשים בהם רק לאחר שינויים רבים.

פרויקט תוכנה גדול הוא מסובך מטבעו. הוא מכיל מגוון רחב של אפשרויות, ולרוב אדם בודד אינו מסוגל לתפוס אותו במלואו. עם זאת, עקב העלות שלו, לא נרצה לזרוק פרויקט גדול, ולכן נגדיר את איכותו של פרויקט גדול על ידי גורמים כגון האפשרות לשימוש חוזר בחלקים מהפרויקט, האפשרות להרחבת הפרויקט והתאימות של הפרויקט למערכות שונות.

תכנות מונחה עצמים מנסה לפשט את הסיבוך של פרויקטים גדולים. על ידי הצגת העולם כאובייקטים, תכנות מונחה עצמים מנסה לתת הפשטה הקרובה יותר לעולם האמיתי. בנוסף, חלוקת העולם למחלקות מנסה לאפשר עבודת צוות טובה יותר על פרויקט – כל אדם מתרכז במחלקות שבאחריותו.
יתרון נוסף בתכנות מונחה עצמים מגיע כאשר רוצים לתחזק את הפרויקט. כידוע, תחזוקה היא חלק הארי בפרויקטי תוכנה. מהסטטיסטיקות, רוב התחזוקה נובעת משינויים בדרישות הלקוח. כעת, כאשר מגיעים שינויים מהלקוח, לרוב שינויים אלו אינם במבנה המחלקות (העצמים במערכת נשארים אותם עצמים, יחסי ההורשה נשארים וכו') אלא באינטרקציה בין האובייקטים השונים. על ידי הורשה, תכנות מונחה עצמים מאפשר לשנות ולהרחיב את התנהגותם של האובייקטים.
בתכנות הפרוצדורלי – התוכנה בנויה סביב הפתרון לבעיה – למשל, האלגוריתמים בו אנו משתמשים. בתכנות מונחה עצמים – התוכנה בנויה סביב העצמים. פתרון הבעיה יכול להשתנות, אולם העצמים נשארים. תכנות מונחה עצמים מנסה להתרכז מסביב לדבר שהכי פחות סביר שישתנה באפליקציה. באותו נושא – מכיוון שרוב השינויים בפרויקטים נובעים מדרישות הלקוח, בזמן האחרון נפוצה גישה חדשה לתכנון וביצוע פרויקטים – במקום לכתוב דרישות מפורטות כפי שהיה נהוג עד כה, כותבים דרישות כלליות בלבד. במהלך התכנות, ותוך כדי התקדמות הפרויקט, מנסחים את הדרישות המפורטות עם הלקוח. הדגש בהתחלת הפרויקט על האובייקטים הקיימים ועל היחסים ביניהם.

2. אובייקטים

2.1. הגדרת אובייקט

אובייקטים הינם אחד מאבני הבניין הבסיסיות של תכנות מונחה עצמים. אובייקט הוא יישות המאופיינת על ידי מצב ועל ידי התנהגות. לדוגמא - לאובייקט "כלב" יש מצב (שם, גזע, צבע) וכן התנהגויות ("נובח", "אוכל"). אובייקטים בשפת תכנות שומרים את המצב שלהם על ידי משתנים. אובייקטים מגדירים את ההתנהגות שלהם על ידי מתודות.

מושג האובייקט: לאובייקטים יש קיום וזהות, ללא קשר לפעולות שניתן להפעיל עליהן. ניתן לזהות ולהבדיל בין שני אובייקטים נתונים.

אם נסכם, **אובייקט** הוא יישות המוגדרת על ידי: מצב, התנהגות ומזהה (ID).

הערה: לעתים, נהוג לכנות אובייקט בשם **מופע (instance)**.

דוגמא נוספת לאובייקט שניתן לממש בשפת תכנות: אובייקט המייצג אדם. לאדם יכול להיות מצב, כגון שם, גיל, תאריך לידה וכדו'. פעולות שהאדם יכול לבצע יכולות להיות הליכה ממקום למקום, דיבור וכו'.

נשים לב שלא כל דבר הוא בהכרח אובייקט. למשל, גיל האדם איננו אובייקט. גיל האדם הוא **מאפיין** של האובייקט אדם.

אובייקט בשפת תכנות מייצג הפשטה של בעייה כלשהי במרחב הבעיה המתאים לה.
ההפשטה הינה זיהוי המאפיינים החיוניים לפתרון הבעיה שהוגדרה, והתעלמות מהרכיבים הלא חיוניים.
לדוגמא: במערכת ניהול עובדים, הפשטה הגיונית תהיה שהאובייקט עובד ישמור את השכר לשעה של העובד. שמירת מידת הנעליים של העובד, כחלק מהאובייקט, תראה במערכת זו הגיונית פחות.
נשים לב כי הפשטה המתאימה למערכת אחת אינה בהכרח מתאימה למערכת שניה, בה הדגשים והדרישות הם שונים.

2.2. אובייקטים בשפת C

שפת C מכילה מספר אלמנטים שניתן להחשיב אותם כאובייקטים, לפי ההגדרה שנתנו לעיל.
נציג כעת אלמנטים אלו. נשים לב שלא כל אלמנט בשפה הוא בהכרח אובייקט. למשל, משתנים בשפת C: יש להם זהות ומצב, אך אין להם התנהגות. לפיכך הם טיפוסים ולא אובייקטים.

2.2.1. פונקציות בעלות משתנים סטאטיים

פונקציה בעלת משתנה סטאטי הינה בעלת התנהגות מסויימת (פעולת הפונקציה). אנחנו מסוגלים לזהות את הפונקציה לפי שמה, ובכך להבדילה מאובייקטים אחרים. כמו כן, המשתנה הסטאטי שבה משמש לשמירת המצב שלה. לפיכך, ניתן להתייחס לפונקציה סטאטית כאל אובייקט.

דוגמא לפונקציה עם משתנה סטאטי:

```
long ID_Generator()
{
      static long iNextID = 0;
      return ++iNextID;
}
```

2.2.2. מודולים

מודול בשפת C יכול להחשב אף הוא לאובייקט. מודול הוא קובץ H בשילוב עם קובץ C, המקיימים כי קובץ ה-H מציע ממשק מסויים למשתמש, קובץ ה-C מממש את הממשק הזה. קובץ ה-C מכיל מספר משתנים סטאטיים שהינם משתנים פנימיים של המודול.

דוגמא למודול: מחסנית המומשת על ידי משתנים סטטיים.

התכונות האובייקטליות של מודול:

- מודול שומר מצב על ידי משתנים סטאטיים.
- מודול הוא אובייקט יחיד – לא יכולים להיות קיימים שני אלמנטים מסוג המודול.
- התנהגות המודול מוגדרת על ידי מספר פונקציות – כל הפונקציות שהוגדרו בקובץ ה-H.

2.2.3. מבני נתונים מופשטים – ADT

מושג ה-ADT הוא המושג הקרוב ביותר לאובייקט אליו ניתן להגיע בשפת C.
תזכורת: ADT הינו רעיון של הגדרת טיפוס ופעולות עליו. אנו מגדירים מבנה (struct) המכיל משתנים, ומגדירים אוסף פונקציות (פעולות) המתאימות למבנה זה. האדם המשתמש ב-ADT משתמש בפונקציות שהגדרנו כדי להתעסק עם משתנים מהסוג החדש.

התכונות האובייקטליות של ADT:

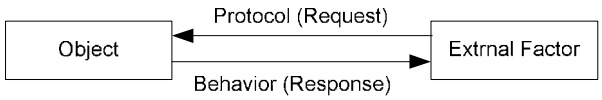
- ניתן ליצור מספר אלמנטים מסוג ADT מסויים. נזהה בין האלמנטים (האובייקטים) השונים, על ידי השוואת הכתובת של כל אחד מהם.
- התנהגות האובייקטים מוגדרת על ידי הפונקציות שהוגדרו ב-ADT.
- מצב כל אובייקט נשמר כמשתנים במופע של ה-struct שהוגדר על ידי ה-ADT.

דוגמא ל-ADT: הסוג FILE הינו struct שלו פונקציות שרות המספקות פעולות: פתיחה, סגירה, כתיבה וכו'.

2.3. מושג האובייקט

נחלק את מושג האובייקט לשלושה נושאים:

- זהות האובייקט. זהות האובייקט מתבטאת ב:
  - מתי העצם קיים (נוצר, נעלם).
  - כיצד הוא יגיב לשאלות השוואה והשמה (שיוויון לאובייקט אחר – מצב או זהות).
- החלק הסטאטי של האובייקט:
  - **מבנה האובייקט**: מאפיינים הניתנים לשינוי/לא ניתנים לשינוי, מאפיינים שהם ערכים, מאפיינים שהם התייחסויות.
  - מצב האובייקט – הערך של השדות שבו. ברגע נתון – הערך הוא קבוע.
- החלק הדינאמי של האובייקט:
  - הפרוטוקול: ההודעות שהאובייקט מכיר.
  - **התנהגות האובייקט**: התגובה של האובייקט להודעות.



2.3.1. החלק הסטאטי של האובייקט

נפרט מעט יותר על המרכיבים הסטאטיים של האובייקט.

מושגים בסיסיים:

- **מאפיינים (Attributes)**: המקום בו מצב האובייקט נשמר. מכונים לעתים: שדות, מאפיינים, תכונות, instance variables, data members.
- **מבנה האובייקט (Structure)**: אוסף כל המאפיינים של האובייקט.
- **מצב האובייקט**: הערך הנוכחי של מאפייני האובייקט השונים.

סוגי מאפיינים:

- **Immutable** - מאפיינים סטאטיים - מאפיינים שאינם ניתנים לשינוי.
- **Mutable** - מאפיינים דינמיים - מאפינים הניתנים לשינוי.

דוגמא:

```
class Person
{
	// Immutable Attribute
	const int IDNumber;

	// Mutable Attribute
	int Age;
};
```

סוגי הערכים שאובייקט יכול לשמור בתוכו:

- **סקלרים**: שלמים, מספרים ממשיים, ערכים בוליאניים.
- **מכולות**: מערכים, רשימות, עצים וכו'.
- **אובייקטים אחרים**.

2.3.1.1. סמנטיקת ערכים מול סמנטיקת התייחסויות

כפי שציינו – אובייקט יכול להכיל אובייקט אחר. נשים לב כי ישנן שתי דרכים לייצוג ערכים מסוג אובייקט: על ידי value או על ידי משתנה התייחסות/מצביע.

סמנטיקת ערכים (VS): האובייקטים נשמרים כחלק מהאובייקט האחר. דוגמא לכך היא ערכים שאינם מצביעים בשפת C++. אם נסתכל על תמונת הזיכרון של האובייקט, אז האובייקט המוכל הינו חלק מתמונת זיכרון זו.
סמנטיקת התייחסויות (RS): האובייקט יכיל התייחסות אל אובייקטים אחרים. האובייקט לא מכיל את האובייקטים עצמם בתוכו. דוגמאות: כל האובייקטים הלא אטומיים בשפת Smalltalk, מצביעים ומשתני התייחסות בשפת C++. האובייקט עצמו אינו מוכל, אלא שדה המאפשר לגשת אל האובייקט המוכל – מצביע אליו.

<u>השוואה בין הגישות</u>:

גישה לאובייקט:	על ידי VS מהירה יותר.
השוואת אובייקטים והעתקת אובייקט:	RS מהיר יותר.
בעלות על אובייקט אחר:	לא מוגדר ב-RS.
שיתוף של אובייקט:	קשה ב-VS.
ניהול זכרון אוטומטי:	קל יותר למימוש ב-RS.

לכל אחת מן הגישות יש יתרון במקרים מסויימים. לפיכך - לא קיימת שפה התומכת רק בסמנטיקת ערכים או רק בסמנטיקת התייחסויות.

<u>חסרונות של שימוש במשתני התייחסות</u>:

- **גישה לא ישירה אל האובייקט**: יותר פקודות בשפת מכונה נדרשות בכל פעם שניגשים לערך.
- **ניהול זיכרון**: המתכנת צריך להתעסק יותר בניהול הזיכרון (במערכות בהן אין ניהול זיכרון אוטומטי).
- **Dynamic binding overhead**: סוג האובייקט ידוע לרוב רק בזמן ריצה, ולכן המהדר יכול פחות לבצע אופטימיזציות בזמן קומפילציה.

2.3.2. זהות האובייקט

זהות: התכונה של אובייקט המבדילה אותו מאחרים. האובייקט יכול להשתנות אולם הזהות שלו נשמרת.
זמן החיים של אובייקט: משך הזמן בין הרגע שהאובייקט נוצרך וצורך זיכרון עד הרגע שהזכרון משוחרר. אובייקט יכול להיות קיים אפילו אם אין שום התייחסות אליו.

קיימים מקרים בהם יש צורך למתודה של האובייקט לגשת אל האובייקט עצמו (לזהות את האובייקט). פעולה זו נעשית על ידי מילים שמורות בשפות השונות. this בשפת C++, self ב-Smalltalk.

2.3.2.1. זהות: שיוויון והשמה

קיימים שני סוגים של שוויון בין אובייקטים:

- **שיוויון התייחסויות**: שתי התייחסויות אל אותו אובייקט בדיוק.
- **שוויון מצב**: שני אובייקטים הנמצאים באותו מצב.

כמו כן, קיימים שני סוגים של השמה:

- **השמת התייחסות**: משתנה ההתייחסות משוכפל. קיים בפועל רק אובייקט אחד. אחרי ההשמה, קיימים שני שמות המתייחסים אל אותו אובייקט.
- **השמה conventional**: המצב של האובייקט מועתק. קיימים בסוף שני אובייקטים המכילים את אותו מצב.

השפות הקיימות תומכות רובן בשוויון והשמה משני הסוגים.

2.3.3. החלק הדינאמי של אובייקט

נחלק את האובייקטים לשני סוגים:

- **אובייקטים פעילים**: מנוהלים על ידי תהליך משלהם (thread). ייתכן שיהיה להם שינויי מצב ספונטניים.
- **אובייקטים פסיביים**: משתנים רק כתוצאת מהפעלה על ידי אובייקטים אחרים. לקוח משתמש באובייקט המספק לו שירותים.

איך לקוח מתקשר עם העצמים השונים? הלקוח: מסוגל לשלוח הודעות לאובייקט כדי לקבל מידע על המצב שלו, או לבקש ממנו לשנות אותו. ללקוח אין גישה ישירה אל מאפייני האובייקט.

העברת הודעות: דרך התקשורת בין הלקוח אל האובייקט.
מינוח בשפת C++: מתודות והודעות מכונות function members בשפת C++.

הלקוח: יכול לשלוח הודעה. הודעה היא שם סימבולי + ארגומנטים במידת הצורך.
אובייקט: מפעיל מתודה בתגובה להודעה, יכול להחזיר במקרה הצורה תשובה ללקוח.

<u>ההבדל בין הודעה לשיטה</u>:
הודעה – פעולה מופשטת שאנחנו רוצים לבצע.
מתודה – המימוש של הפעולה.

דוגמא להודעה היא אפילו האופרטור +.
משתנים מסוגים שונים יתנהגו בצורה שונה כאשר נחבר ביניהם. יתר על כן – בשפת C++ ניתן לחפוף אופרטורים ולהגדיר אפילו עבור מחלקות שאנחנו כותבים את האופרטור, ובכך לראות בצורה טובה את ההבדל בין ההודעה "+" לפעולה, המתודה, המבוצעת בהתאם.

נגיד שולחים לאובייקט הודעה, אם איננו יודעים מאיזה סוג הוא, איננו יודעים (ולא מעניין אותנו) איזה מתודה תבוצע.

היכולת להפריד בין ההודעה למתודה מושג על ידי תכונת הפולימורפיזם בתכנות מונחה עצמים.

ניקח לדוגמא את האובייקט צורה ואת הפעולה סיבוב.

עבור צורה כללית, אם הזווית היא 0° לא נעשה כלום, ואחרת נסובב אותה בזווית המבוקשת.

עבור עיגול, עבור כל זווית אין צורך לבצע כל פעולה.

עבור מלבן, סיבוב ב-180° או ב-360° אינו מצריך פעולה.

כל אחד מהאובייקטים יגיב בצורה שונה להודעה "סובב את עצמך".

2.3.3.1. שליחת הודעות מול קריאה לפונקציות

	שליחת הודעה	קריאה לפונקציה
מקבל	אובייקט יעד האחראי לביצוע הפעולה המבוקשת. תמיד חייב להיות מישהו שיקבל את ההודעה.	לא תמיד קיים מקבל להודעה. למשל במקרה של fork() או exit().
קישור	לא קשור לגוף פונקציה יחיד בהכרח. המימוש יכול להיות שונה בהתאם למקבל. יתכן שהמקבל לא ידוע עד לזמן הריצה.	שם הפונקציה מקושר לגוף פרוצדורה מסויים בצורה חזקה.
סינכרוניזציה	הקריאות יכולות להיות א-סינכרוניות.	הקריאות הן לרוב סינכרוניות.

התנהגות: מה שהעצמים (והתוכנה) עושים. כולל שינויי מצב פנימיים, הודעות הנשלחות אל אובייקטים אחרים והערך אותה הפעולה מחזירה.

פרוטוקול: הדרך להשיג את ההתנהגות. אנקפסולציה של מושג ההתנהגות

2.3.3.2. סוגי פעולות (אופרטורים) על אובייקטים

1. **mutator** – אופרטורים שהתפקיד שלהם הוא לשנות את מצב האובייקט.
2. **inspector** – אופטורים שהתפקיד שלהם הוא לדווח על מצב העצם. Inspector מחזיר אובייקט שונה עם עותק של המידע הקיים בעצם הנצפה.

- קיימת פונקציה המשנה מצב, וגם מחזירה את הערך הקודם של האובייקט. פונקציה זו במקרים רבים היא יעילה יותר. מבחינת הגישה, היא פחות בגישה של תכנות מונחה עצמים אבל שימוש בפונקציות כאלו לא נחשב לרוב תכנות גרוע.

3. **constructor** – הודעה שבד"כ לא הולכת אל העצם אלא אל המחלקה. ההודעה מיועדת ליצור אובייקט חדש ולאתחל את השדות שלו.
4. **destructor** – מנקה את האובייקט ומבצע פעולות סיום נוספות. הפעולה היא פעולה על העצם (לא על המחלקה).

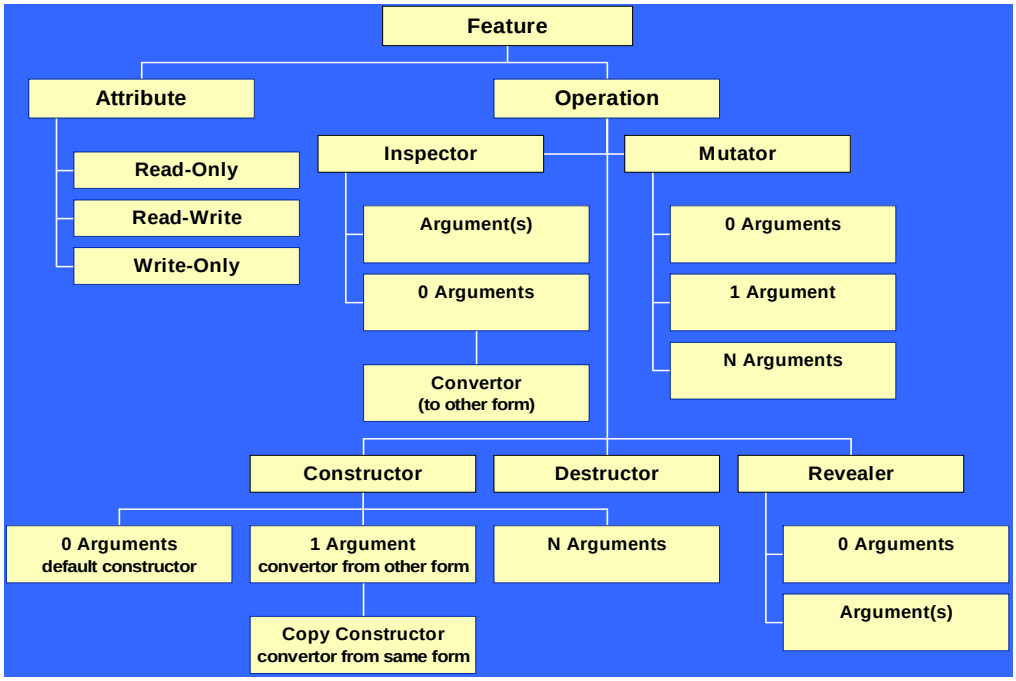
- נשים לב שהעובדה ש-destructor זו הודעה לעצם ו-constructor זוהי הודעה לאובייקט, זוהי הסיבה שבשפת C++ ה-destructor יכול להיות וירטואלי וה-constructor לא.

5. **revealer** – נותנת גישה לחלקי האובייקט הפנימיים. למשל, מחזירה משתנה התייחסות אל משתנה פנימי.

נציג דוגמא הכתובה בשפת C++. הדוגמא לקוחה מהרצאות הטכניון בקורס "תכנות מונחה עצמים":

```
class Date {
public:
// Constructor
      Date(int day = -1, int month = -1, int year = -1);
// Inspectors
      int day(void) const { return d; }
      int month(void) const { return m; }
      int year(void) const { return y; }
// Mutators
      int day(int day);
      int month(int month);
      int year(int year);
// Destructor
       ~Date(void) { cout << "Destructor Code Here" << endl; }
private:
      int d;
      int m;
      int y;
};
```

השרטוט הבא מדגים את רכיבי הפרוטוקול השונים הקיימים:



- Inspector עם 0 ארגומנטים – הפעולה נקבעת על פי שם המתודה. למשל: GetName.
  Inspector עם 0 ארגומנטים זהו ייצוג ל-read only attribute.
- באופן מקביל, mutator עם ארגומנט 1 מייצג write-only attribute.

שפות המציעות תכנות מונחה עצמים, צריכות לאפשר הסתרת חלקים מהמימוש. כאשר שפה אינה מאפשרת להסתיר חלקים מהמימוש, זו בעייה בקונספט של תכנות מונחה עצמים.
איננו רוצים בתכנות מונחה עצמים לאפשר למשתמש גישה ישירה אל ה-data members.

עקרון ההתייחסות האחידה (The Principle of Uniform Reference) אומר שהתייחסות לאופציה כלשהי שמספק האובייקט לא צריכה להיות שונה במקרה שמדובר במאפיין או בפעולה. ב-C++ דבר זה לא מתקיים. קיים הבדל אם נקרא לפונקציה, למשל myClass.x() לבין אם ניגש לשדה myClass.x.

לפי עקרון הסתרת המימוש ב-OOP, שדות נמצאים ב-C++ כ-private, ואנו מספקים פונקציות קטנות – get ו-set לגישה אליהם. על מנת שלא לפגוע בביצועי התוכנית בקריאה לפונקציות קטנות רבות הוכנסה המילה inline ל-C. שימוש בה מביא לשיפורים של פי 25 לעומת תוכנית זהה שלא משתמשת בה!

3. מחלקות

מחלקה היא הפשטה על עצמים. היא מנסה לאגד קבוצות דומות של עצמים סביב הפשטה אחת. מחלקה מייצגת קבוצת אובייקטים החולקים מבנה משותף והתנהגות משותפת.

3.1. מהי מחלקה

הפשטה מעל אובייקטים

קבוצת אובייקטים החולקת:

- אספקטים דינאמיים: פרוטוקול והתנהגות.
- אספקטים סטאטיים: מבנה האובייקט (אבל לא בהכרח את הערכים של השדות השונים).

תבניות ליצירת אובייקטים

מחלקה משמשת ליצירת אובייקטים עם זהות שונה, להם פרוטוקול, התנהגות ומבנה זהה, אולם יתכן שמצב שונה.

בניגוד לאובייקט קונקרטי, מחלקה לא בהכרח קיימת בזמן ריצה ובמרחב הזיכרון של התוכנית.

אלמנטים שאינם מחלקות

אובייקט הוא איננו מחלקה, אם כי מחלקה עשויה להיות אובייקט. למשל, קיימות שפות בהן הדרך היחידה שבה נוצרים אובייקטים חדשים היא על ידי שאובייקטים קיימים מאתחלים אותם.

כמו כן, לא כל קבוצת אובייקטים יכולה להיות מחלקה בודדת.

3.2. הקשר בין ADT למחלקות

טיפוס – אוסף עצמים שאנחנו מפרשים בצורה אחידה. על קבוצת עצמים מסוג מסוים ניתן לקבוע פעולות מותרות ואסורות.

ADT – הפשטה על טיפוס – לא חייבים לציין את הטיפוס ומתרכזים בפעולות עליו.

מחלקה מבצעת צעד נוסף: יש עצמים שיש להם טיפוס ולכן יש פעולות המותרות עליהם. עם זאת יש להם מנגנונים נוספים ושירותים נוספים. למשל – מחלקה יודעת לנהל יותר טוב את השגיאות בה.

3.3. ניהול מחלקות

למה צריך לנהל מחלקות?
כשאנחנו מריצים את התוכנית, מי שפועל הם העצמים, ולא המחלקות שיצרו אותם. המחלקות לכאורה אינן חלק מהדיון בזמן ריצה. אולם, עדיין צריך להכיר את המחלקות עקב תכונת הרב צורתיות – פולימורפיזם. נרצה ששליחת הודעה לאובייקט תפעיל את המתודה המתאימה לה. כדי לדעת איזה מתודה צריך להפעיל בכל מקרה, אנו צריכים לדעת לאיזו מחלקה שייך האובייקט, ומכאן כן יש צורך במידע מסויים על המחלקות בזמן ריצה.

אם מושג המחלקה הינו עצם, אז ניתן לקבל בזמן ריצה את המידע הדרוש.

3.3.1. האם נרצה שמחלקה תהיה אובייקט?

מחלקה – אחד המושגים החשובים ביותר של התכנות מונחה העצמים.
האם היינו רוצים להפוך מחלקה לאובייקט? איזה סיבות יכולות להיות לכך?

- **ctor** – מחלקה היא תבנית ליצירת אובייקטים. מי ייצור את האובייקט החדשה? ניתן לחשוב שאובייקט מקבל הודעה ונולד – אולם לפי גישה זו, האובייקט מקבל הודעה עוד לפני שהוא נוצר. יותר טבעי לחשוב כאילו מחלקה היא אובייקט – והיא תקבל הודעה לייצור אובייקט חדש.
- **Dynamic dispatch** – הודעה m מועברת לאובייקט – בתלות האובייקט צריך להגיב. צריך לקבוע מי מחשב עבור m + אובייקט איזו מתודה תופעל. התנהגות ופרוטוקול משותפים לכל האובייקטים של אותה המחלקה - ולכן נרצה שה-dispatcher יהיה במחלקה – במקום אחד לכל אובייקט שבזיכרון.
- **Garbage allocation** – בדומה להודעה – מגיע למקום בזיכרון וצריך לדעת מה נמצא שם. יש טעם שהמחלקה שלהם ("המפה שלהם") תהיה במקום מסוים בזיכרון – ולכן יש טעם שהמחלקה תהיה אובייקט.

- **save to file ,clone ,deepcopy** – פעולות אלו צריכות לאתר את מבנה האובייקט ("מפה"). המבנה משותף לכל האובייקטים של אותה מחלקה ולכן יהיה שימושי גם במקרה זה לשמור את המחלקה כאובייקט.

מכל נקודות אלו נראה כי יש יתרונות אם נבחר לייצג מחלקה בתור אובייקט.

מתעוררת השאלה: לאיזו מחלקה שייך אובייקט מסוג מחלקה?

שאלה נוספת: אם מחלקות הן עצמים, אז למה שלא יהיה ניתן ליצור מחלקות חדשות בזמן ריצה?

Meta Classes .3.3.2

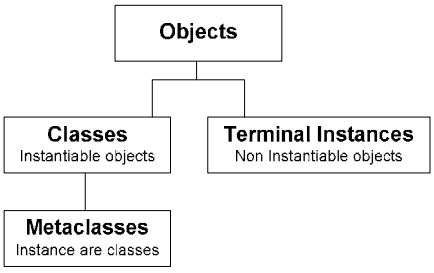
נתייחס אל כל העולם כאל עצמים.

חלק מהעצמים הינם עצמים שרק פועלים – הם אינם יכולים ליצור עצמים חדשים.

עצמים אחרים – מחלקות – אחראיים על קבוצת העצמים, יוצרים מחלקות.

חלוקה לקטגוריות:

- **מחלקה**: אובייקט המייצר עצמים.
- **מטה-מחלקה (metaclass)**: מחלקה שהעצמים שהיא מייצרת הם מחלקות.
- **Terminal Instance**: עצמים שמהם לא יוצרים עצמים נוספים.



הרעיון ניתן לפיתוח הלאה – האם ישנן מחלקות המייצרות מטה-מחלקות? וכו'. ישנם מודלים שונים, הנבדלים ביניהם במספר "הרמות" של מטה מחלקות.

רוב השפות יהיו עם בין 1-4 רמות, אולם באופן תיאורטי יכולים להיות ∞ רמות שונות.

The 1 Level System .3.3.2.1

לפי גישה זו אין כלל מחלקות בשפה. קיימים עצמים היודעים לשכפל את עצמם. שפה התומכת בגישה זו צריכה לספק מנגנון שאינו קשור למודל, על מנת שנוכל לייצר את העצמים הראשוניים.

לכל אובייקט מסוג X יש Prototype המגדיר כיצד ליצור איברים מסוג X.
כשאנו רוצים ליצור אובייקטים חדשים מסוג X אנו עושים זאת על ידי מילוי שדות ריקים ב-Prototype.

שפה לדוגמא שיש בה 1-Level היא Self.

The 2 Level System .3.3.2.2

הפרדה בין מחלקות לעצמים (C++).
ברמה נמוכה העצמים – חיים בזמן ריצה.
מעליהם המחלקות – המחלקות אינן קיימות בזמן ריצה. המחלקות הינן עצמים רק בזמן הקומפילציה.

The 3 Level System .3.3.2.3

במודל 3 רמות מחלקה הינה גם עצם. קיימת מחלקה class שהיא meta class – משמשת לתיאור ויצירת מחלקות. כל המחלקות הינן עצמים מסוג class.

במודל 3 רמות ניתן לתת פקודה למחלקה class כדי שתיצור מחלקה חדשה (במודל 2 רמות לא ניתן לעשות זאת).

למבנה 3 רמות יש חסרון שלא קיים ב-2 רמות:static functions, static members לא משתלבים במודל זה. אם מחלקה צריכה להכיל פונקציה סטטית, עובדה זו צריכה להופיע בתיאור המחלקה, אולם המחלקה class לא יכולה לתאר אבחנה זו כי היא אחראית לתאר את כל המחלקות (נשים לב שכל מחלקה היא instance של המחלקה class, ולא מחלקה נורשת ממנה!). במחלקה class מופיע מה צריך להיות בכל מחלקה.

במודל 3 רמות:

- המחלקה Class נורשת מהמחלקה Object. המחלקה Class היא instance של עצמה.
- המחלקה Object היא instance של המחלקה Class, והיא איננה נורשת מאף מחלקה.
- כל מחלקה (שאינה נורשת ממחלקה אחרת), למשל MyClass, נורשת מהמחלקה Object והיא instance של המחלקה Class.
- אובייקטים הינם instance של המחלקות המתאימות להם.

3.3.2.4. מודל 4 רמות

מודל זה דומה למודל 3 רמות, אבל מכיל בנוסף ל-Meta Class מחלקה נוספת – Meta-Meta Class.. הרעיון במודל זה: Meta-Meta Class היא מחלקה המייצרת מטה-מחלקות. מטה-מחלקה היא סינגלטון המגדיר כיצד יוצרים מחלקה מסוימת.

דוגמא תבהיר את הנושא: נניח שהעולם שלנו מורכב מאובייקטים השייכים למחלקה Person. אז:

- אובייקטים השייכים ל-Person הינם instance של המחלקה Person.
- המחלקה Person היא instance של המחלקה Meta-Person, והיא נורשת מ-Object.
- המחלקה Meta Person נורשת מהמחלקה Meta-Meta Class והיא גם Instance שלה.
  מחלקה זו הינה סינגלטון – קיים רק אובייקט אחד שלה – המחלקה עצמה. המחלקה Meta Person מחזיקה אילו משתנים סטטיים וכדו' יהיו במחלקה Person. מתעוררת השאלה: למה להחזיק משתנים סטטיים ב-Meta Person ולא במחלקה Person עצמה?
  התשובה: המשתנים כן יהיו במחלקה Person. במחלקה Meta Person רק מוגדר כיצד יש לייצר את המחלקה Person.
- Meta-Meta class היא instance של עצמה, והיא יורשת מ-Object.

4. הורשה – Strict Inheritance

4.1. מבוא

הדיון שלנו על הורשה יתחיל מהורשה פשוטה יחסית, ולאחר מכן נציג הורשה שהינה מתוחכמת יותר, אולם גם מעניקה לנו המתכנתים יותר כוח.

אנחנו בד"כ נזהה את ההורשה הפשוטה במצב בו קיימת מחלקה עם עצמים היודעים לבצע פעולה כלשהי, וכן קיימת קבוצת עצמים נוספת היודעים לבצע פעולה זו וקצת יותר. מכאן, בד"כ אנו מגלים את הצורך בהורשה כאשר אנחנו רואים קוד / פונקציות משותפות בשתי מחלקות.
מכאן, השימוש הראשון של הורשה הינו: **שימוש בהורשה למניעת שכפול קוד**.
למרות שמניעת שכפול קוד היא שימוש חשוב, זהו איננו השימוש החשוב ביותר של הורשה. נציג מיד שימושים נוספים ליכולת ההורשה.

אפשרות נוספת שהורשה נותנת לנו היא ארגון עצמים בעולם בצורה היררכית, על ידי שימוש בהפשטות מתאימות.

בשפות שונות ישנם מונחים שונים המתאימים למושגים של תכנות מונחה עצמים. נציג מעט טרמינולוגיה, השוואה בין מונחים בשפת C++ למונחים בשפת Little Smalltalk:

Smalltalk	C++
Inherit	Inherit/Derive
Super Class	Base class
Subclass	Derived class
Instance Variable	Data Member
Method	Member Function
Message	Member Function Call
Class Variable	Static Data Member
Class Method	Static Function Member

4.2. מערכות סוגים ו-Strict Inheritance

4.2.1. סיווג שפות

מספר מושגים:

- **ערך (value)**: אוסף ביטים המייצג מצב של עצמים.
- **סוג (type)**: משמעות לאוסף הביטים המהווים ערך. ניתן להגיד שסוג זהו מנגנון לפירוש המשמעות של ערך. בנוסף – טיפוס מגדיר ערכים שהם חוקיים וערכים שהם לא חוקיים עבורו.
- **משתנה (variable)**: שם לתא בזיכרון שיכול להתקין ערך.

למה בעצם אנחנו צריכים לדבר על ערכים וסוגים כאשר אנחנו מדברים על הורשה?
נניח שיש לנו את הביטוי a + b. נשאל: איזה פעולה תבוצע? אם a, b הינם שלמים, יתבצע חיבור בשלמים. אם a, b הם מספרים מסוג float יתבצע חישוב נקודה צפה. אם הם מחלקות, ייקרא ה-operator+ שהוגדר עבורם.
לפיכך, יש חשיבות לדעת מה הסוג של האובייקט כדי לדעת איזה פעולה תפעל עליו, ולכן נפתח את הדיון בנושא.

סיווג שפות לפי חשיבות הסוגים:

- **Strongly Typed Languages**: הסוג מחובר חזק אל הערך. לא ניתן לשבור את הקשר במסגרת השפה. דוגמאות לשפות: ML, Eiffel, Modula.
- **Weakly Typed Languages**: לערכים יש סוגים מתאימים, אבל המתכנת יכול לשבור או להתעלם מהסוג. שפות: C, Turbo-Pascal.
- **Untyped Languages**: לערכים אין סוג מוגדר. שפות לדוגמא: Assembly, Lisp, ...

סיווג שפות לפי זמן אכיפת הסוגים:

- **Dynamic Typing**: חוקי הסוגים נאכפים בזמן ריצה. למשתנים אין סוג המשוייך אליהם. דוגמא: PROLOG, Smalltalk.
- **Static Typing**: חוקי הסוגים נאכפים בזמן ההידור. לכל משתנה קיים סוג המתאים לו. שפות: C, פקסל, Eiffel, ML.

כפי שצוין, לשפת C טיפוסיות חלשה. ניתן לעשות casting ולהתייחס לכל נתונים שהם כאל מערך של בתים. בדומה, גם union הקיים בשפת C הינו סימן לטיפוסיות חלשה.
בשפת C אפילו הביטוי הבא חוקי, וזאת כי הסוגריים הופכים לפעולת החיבור:

```
int *p;
3[p];
```

האם טיפוסיות חלשה היא בהכרח דבר רע? יש לטיפוסיות חלשה שימושים לא מעטים. למשל, שימוש שנעשה לעיתים הינו ליצר רשימה מקושרת חסכונית בזיכרון – על ידי שימוש ב-XOR בין כתובת ה-NEXT לכתובת הקודמת, ובכך להשיג רשימה דו כיוונית בעלות של רשימה חד כיוונית.
בעיה עם המימוש של רשימה מקושרת דו כיוונית עם XOR בין מצביעים: אנחנו מסתמכים בה על ההנחה של-long ול-address יש גודל זהה. אם בעתיד עובדה זו תשתנה, כל הרשימות המבוססות על טריק זה יפסיקו לעבוד.

לפיכך – לפי הגישה של OOP, המנסה להפריד בין המימוש לבין הממשק, טיפוסיות חלשה היא בהחלט דבר לא חיובי.

4.2.2. Static Strong Typing

Static Typing הולך כמעט תמיד עם Strongly Typing. במקרים רבים מתייחסים אליהם באופן סימטרי ואף מחליפים בין השמות שלהם. השם שתכנות מונחה עצמים מעדיף הוא Strong Typing, מכיוון שכפי שנראה מאוחר יותר יש צורת כתיבה ל-Dynamic Binding גם בשפות שהן Strongly Typed System.

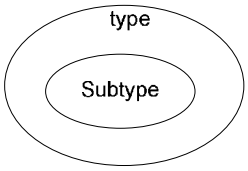
מדוע נרצה להשתמש ב-Strong Typing?

א. **בטיחות** – מובטח שתוכנית שהיא st לא תעשה בזמן ריצה שגיאות של טיפוסים (שגיאה בה אובייקט לא מסוגל לטפל בהודעה הנשלחת אליו מכיוון שהוא לא מכיר אותה)! עבור שפות שהן dynamic typed, בזמן ריצה המערכת צריכה לבצע בדיקת טיפוסים.
ב. **ביצועים** – מערכת Static Strong Typing תעבוד יותר מהר כי מובטח שבזמן ריצה אין צורך לבדוק שגיאות טיפוסים, ולכן אנחנו חוסכים פעולות..
ג. **בהירות** – st – אמירות שנותנות רמיזות לקורא מה התוכנית עושה על ידי הצהרה על הסוגים השונים של משתנים. עובדה זו עוזרת בתוכניות גדולות.

כל היתרונות החשובים הללו מתבטאים במערכות גדולות. במערכת קטנה שתרוץ פעם אחת – נושא זה הינו פחות משמעותי.

4.2.3. Strict Inheritance

Strict Inheritance זהו סוג ההורשה הפשוט ביותר, ומשמעותו – הורשה המרחיבה את מחלקת הבסיס. המחלקה החדשה יכולה להרחיב את המבנה, הפרוטוקול וההתנהגות של מחלקת האב. בסוג זה של הורשה המחלקה היורשת אינה משנה את ההתנהגות המקורית של האב אלא רק מרחיבה אותה.



בהורשה מסוג זה, כל אובייקט השייך לסוג הנורש הוא גם אובייקט מהסוג הראשון. מבחינת תורת הקבוצות, היחס בין סוג הבסיס לסוג הנורש ממנו הינו יחס של הכלה.

היחס בין הבן לאב הוא יחס Is-A: אובייקט של מחלקת הבן הוא גם אובייקט של מחלקת האב. לפיכך, כל פונקציה המקבלת אובייקט מסוג האב, תהיה מסוגלת לקבל גם אובייקט מסוג הבן.

אנחנו מקבלים כאן את השימוש החשוב השני של ההורשה: **ממשק אחיד**. הממשק מוגדר במחלקת הבסיס וקיים אצל כל הבנים היורשים ממחלקה זו.

4.2.4. סוגים ותכנות מונחה עצמים

טיפוס ומחלקה אינם תמיד מושג זהה. בשפת LST טיפוס ומחלקה הינם מונחים שקולים.
בשפת C++ לעומת זאת מחלקה היא אלמנט הרבה יותר חזק – מטיפוס בשפת C++ לא ניתן לרשת.

הבדל בין מחלקה לטיפוס: מחלקה יכולה לנהל בצורה מלאה את התנהגות העצם.
למשל, חריגה ב-int לא מתבטאת ב-C (חלוקה ב-0 לעומת זו כן).
לעומת זאת מחלקה יכולה להגדיר את התנהגותה באופן מוחלט עבור כל מקרה.

חלק חשוב בהורשה הוא **תאימות**. כל פעולה שנעשית על האב, חייבת לפעול נכון אצל כל בן. מכיוון שכל פונקציה שמקבלת את האב יכולה לקבל גם אובייקט מסוג הבן, הבן חייב לשמור על המוסכמות בהם עומד האב.

ל-strict inheritance יש חסרון אחד מרכזי, והוא שאי אפשר לעשות באמצעותו הרבה – מכיוון שלא ניתן לשנות את התנהגות האב. לפיכך, נציג מיד סוגי הורשה נוספים.

4.3. פולימורפיזם

4.3.1. מבוא

סוג חזק יותר של הורשה הינו רב צורתיות. הרעיון של רב צורתיות הינו התנהגות שונה.

שורת קוד פולימורפית: אותה שורת קוד יכולה לעשות דברים שונים במצבים שונים.

מתי נרצה פולימורפיזם? אלגוריתמים כלליים. לדוגמא:

חיפוש איבר X: נתחיל באיבר הראשון. כל זמן שלא הגענו לסוף המכולה – עבור לאיבר הבא. אם מצאת את האיבר המבוקש, החזר אמת. אחרת המשך.

אלגוריתם זה הוא אלגוריתם פולימורפי. אותו קוד של אלגוריתם החיפוש יכול להתאים לאובייקטים מסוגים שונים.

בשפות Dynamically typed כל שורות הקוד הן פולימורפיות (כמעט לפי הגדרה).

בשפות statically typed הקוד מוגבל למשתנים מהסוג שהוגדר בזמן ההידור.

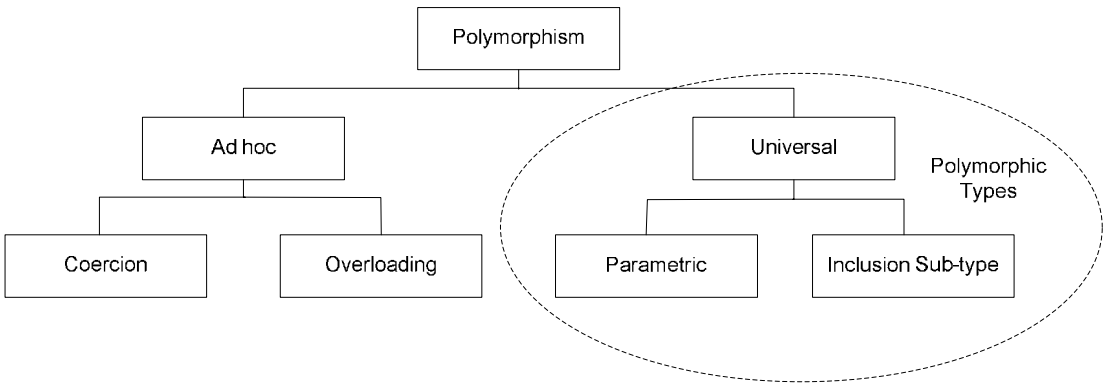
האתגר: השגת פולימורפיות בשפות שהן statically typed, ועל ידי כך להשיג: יכולת ביטוי מורחבת וגם בטיחות בריצת התוכנית.

4.3.2. סוגי פולימורפיזם

4.3.2.1. תקציר

סוגי הפולימורפיזם השונים:

- **overloading**: מזהה יחיד מייצג מספר אבסטרקטיות בו זמנית.
למשל: operator overloading. שפת C++ מגדירה overloading לאופרטור + עבור שלמים ועבור משתני float, אבל אם לא נגדיר במפורש, עבור המחלקה Person לא יהיה קיים האופרטור +.
- **coercion**: אבסטרקציה יחידה מסוגלת לשרת מספר סוגים על ידי כפייה מרומזת (הרחבת השימוש באבסטרקציה יחידה על ידי המרות סוגים).
דוגמא: המרת טיפוסים: int ל-double וגם double ל-int.
- **parametric**: אבסטרקציות הפועלות בצורה אחידה על ערכים מסוגים שונים.
לדוגמא: template בשפת C++.
- **inheritance**: תת סוגים היורשים פעולות מסוג ההורה.



Universal Polymorphism מול Ad Hoc Polymorphism .4.3.2.2

Universal

- הפולימורפיזם הוא מעל מספר אינסופי של סוגים
- הצורות השונות של הפולימורפיזם נוצרות באופן אוטומטי.
- קיים בסיס אחד משותף ואחיד שממנו כל הצורות של הפולימורפיזם לוקחות צורה.

Ad hoc

- פולימורפיזם מעל מספר סופי של צורות – לרוב, מספר צורות בודדות.
- צורות שונות של פולימורפיזם מקודדות באופן ידני – או פסדו-ידני.

אין בסיס משותף אחד לכל צורות הפולימורפיזם, מלבד זה שבכוונת המתכנת. צורות שונות יכולות להתנהג בצורה שונה.

Ad Hoc Polymorphism .4.3.2.3

Overloading .4.3.2.3.1

העמסה: נתינה של יותר ממשמעות אחת למושג. המשמעויות השונות יכולות להיות, אולם אינן חייבות, להיות קשורות.
דוגמא למשל: למילה static ב-C/C++ יש מספר משמעויות, בהתאם להקשר בו אנחנו משתמשים בה.

נאמר ש**אופרטור הוא מועמס** אם האופרטור מייצג שתיים או יותר פונקציות שונות.
ב-C ,Pascal ו-ML רק אופרטורים ומזהים הנמצאים build-in בשפה הם אופרטורים מועמסים.
C++ לעומתם מכילה מנגנון המאפשר למתכנת לחפוף אופרטורים (כמעט) כרצונו.

Coercions .4.3.2.3.2

Coercion זהו מיפוי שקוף של ערכים מסוג אחד אל ערכים מסוג אחר.
לדוגמא, Pascal מבצעת מיפוי שקוף של ערכים מסוג integer לסוג real.
הפונקציה sqrt(n) המיועדת לערכים מסוג real תעבוד גם כאשר n הוא מסוג integer.

שפות מודרניות מנסות לצמצם או להימנע לגמרי מהשימוש בתכונה זו.

השימוש ב-Coercion יוצר לעתים דו משמעות. לעתים נדרשות מספר המרות כדי להמיר מסוג אחד לאחר. המסלול של ההמרה יכול להשפיע על התוצאה הסופית.
מסלול ההמרה אינו עץ ואף אינו בהכרח גרף מכוון.

המצב מסתבך עוד יותר כאשר מערבים coercion עם overloading. מה יקרה, למשל, כאשר מנסים לחבר ערכים מסוגים שונים?
ב-C++ למשל, מיושמת הגישה הבאה:
בהינתן פונקציה F(a1, a2, …, an) יתכנו מספר רב של גרסאות מועמסות של F.
C++ בוחרת בזמן הידור את הפונקציה המתאימה, לפי פרמטרים שונים:

- מקרים בהם אין צורך בהמרה או קיים צורך בלתי נמנע בהמרה:
  $\text{array} \rightarrow \text{pointer}, \text{T} \rightarrow \text{const T}$.
- הגדלת גודל הערך: $\text{short} \rightarrow \text{int}, \text{float} \rightarrow \text{double}$.
- המרות סטנדרטיות: $\text{double} \rightarrow \text{int}, \text{int} \rightarrow \text{double}, \text{derived} \rightarrow \text{base}, \ldots$.
- המרות שהוגדרו על ידי המשתמש.

הבחירה בין האפשרויות השונות נעשית על ידי "תחרות" בין הפונקציות.
המנצח חייב להיות: מתאים יותר מהאחרים בלפחות פרמטר אחד, ולפחות טוב כמו האחרים בשאר הפרמטרים. אם לא קיים מנצח, C++ תדווח על שגיאה.

Universal Polymorphism .4.3.2.4

Parametric Polymorphism .4.3.2.4.1

פולימורפיזם זה קורה עבור מספר אינסופי של סוגים קשורים. פולימורפיזם זה מאפשר להגדיר פעולה מופשטת שתפעל באופן אחיד על ארגומנטים שכולם ממשפחת סוגים דומה.

<u>Ad hoc לעומת Parametric:</u>

Overloading: פעולה מינימלית – למספר קטן של הפשטות קיים מזהה זהה.
אינו מגביר את יכולת הביטוי של השפה – ניתן להיפטר מ-overloading על ידי נתינת שמות שונות להפשטות השונות המתאימות לו. הקשר בין ההפשטות השונות לא בהכרח קיים.
Coercion: פעולה גדולה יותר – אותה פונקציה יכולה לשמש למספר מטרות. עם זאת מספר המטרות הינו מוגבל, והערך המוחזר הינו זהה. הקשר בין הצורות מוכתב על ידי ה-coercions, וקשר זה אינו חלק מהשגרה.
(Polymorphic Type (universal – להפשטה יחידה יש מספר גדול של משפחות של סוגים קשורים. ההפשטה פועלת באופן אחיד על הארגומנטים שלה ללא תלות בסוגם. מוסיפה כוח ביטוי לשפה על ידי יצירת פונקציות היכולות לקבל מספר לא מוגבל של סוגים.

Parametric Type Polymorphism In Pascal: האופרטורים - ,+, *, פעולות איחוד וחיתוך קבוצות הינם כולם Parametric Type Polymorphism. הפרוצדורות new, dispose יוצרות משתנה מסוג כלשהו. הערך nil הוא ערך מצביע לסוג כלשהו. [] הקבוצה הריקה מתאימה לכל סוג.
Universal pointer in C: בשפת C מצביע מסוג *void יכול להיות מושם לתוך כל מצביע אחר, וכן כל מצביע יכול להיות מושם בתוך משתנה מסוג *void. עובדה זו ב-C איננה ad hoc אלא parametric polymorphism, מכיוון שהיא מוגדרת אוטומטית לכל מצביע.
C++ Templates: Templates זוהי הדרך של שפת C++ לממש parametric polymorphism. בעזרת התבניות אנחנו מסוגלים ליצור פונקציות/מחלקות המקבלות פרמטר מסוג כלשהו שיוגדר בהמשך, לפי הצורך.

Inclusion Polymorphism .4.3.2.4.2

Inclusion Polymorphism זהו הסוג השני של Universal Polymorphism. סוג זה נובע מקשרי הכללה (הורשה) בין סוגים או קבוצות של ערכים.

<u>תזכורת</u>:

סוג הוא קבוצת ערכים להם פעולות משותפות. **תת סוג (sub type)** הוא תת קבוצה של ערכים.

דוגמאות:

- "מכוניות" הן תת סוג של "כלי רכב".
- "סטודנטים" ו-"מרצים" הם תת סוג של "בני אדם".

:Subtype

- הגדרה 1: הסוג A הוא תת סוג של B אם $A \subseteq B$.
- הגדרה 2: הסוג A הוא תת סוג של B אם כל ערך ב-A יכול להיות מומר (coerced) לערך ב-B.
- דגש: תת סוג איננו סוג – ערך יכול להיות שייך לסוג אחד בלבד, אולם ערך יכול להיות שייך למספר תתי סוגים.

:Subclassing

- המטרה: שימוש חוזר בקוד.
- זהו מנגנון של מימוש.
- ניתן לומר שזה "תכנות אינקרמנטלי".

ב-Strict Inheritance, מתקיים כי Subclassing $\Leftarrow$ Subtyping. פעולות המוגדרות על מחלקת הבסיס מותרות על המחלקות הנורשות ממנה.

רוב ה-Inclusion Polymorphism נובעים מ-subtypes.

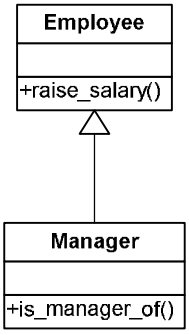
כאשר אנחנו מדברים על Inclusion Polymorphism, אנחנו מדברים על שני אלמנטים:

- מתודות פולימורפיות.
- אובייקטים פולימורפיים.

דוגמא למתודה פולימורפית במחלקות:

```
Employee E;
Manager M;

E.raise_salary(10);      // OK
M.raise_salary(10);      // OK
E.is_manager_of(...);    // Error
E.is_manager_of(10);     // OK
```



המתודה raise_salary היא פולימורפית – היא מסוגלת לקבל את כל תתי הסוגים (תתי המחלקות) של המחלקה Employee.

דוגמאות לאובייקטים פולימורפיים:

- **Pascal**: הערך nil שייך לכל סוגי המצביעים.
- **C**: הערך 0 הוא פולימורפי – שייך לכל סוגי המצביעים.
- **C++**: הסוג *void הוא סוג-על לכל סוגי המצביעים. this הוא אובייקט פולימורפי – יכול להצביע לסוגים שונים בזמנים שונים.
- **Smalltalk**: כל המשתנים בשפה הינם פולימורפיים מכיוון שהם יכולים להכיל ערכים מסוגים שונים בזמנים שונים.

Upcasting, Downcasting .4.3.3

Casting (המרה): המרת coercion של אובייקט ממחלקה נגזרת לאובייקט ממחלקת הבסיס.

Up-casting: המרת מצביעים במעלה היררכית ההורשה (למשל, המרת מצביע מסוג עובד למצביע מסוג בן אדם).

Up-casting של this מתבצע באופן אוטומטי בכל פעם שאנחנו משתמשים במתודה שנורשה ממחלקה אחרת.

```
Emploee E;
Manager M;

M.is_manager_of(E)      // Type of this is Manager*,
                  // No casting occurs

M.raise_salary(10);     // Type of this is Employee*
                  // in raise_salary.
                  // Up casting must occur
```

Down-casting: המרת מצביעים במורד גרף ההורשה. המרה זו חייבת להתבצע על ידי הוראה מפורטת. המרה זו נעשית רק במקרים מיוחדים – זו אינה המרה שנרצה לרוב לבצע.

```
Employee  E, *pE;
Manager   M, *pM;
pE = &M;                // OK: implicit upcasting.
M = *pE;                // Error: implicit downcasting is not allowed

// explicit down casting:
M = *(Manager *)pE;                 // deprecated syntax
M = *static_cast<Manager*>pE;       // recommended syntax
```

נשים לב למה שכתוב בקוד – מומלץ להשתמש בהמרה על ידי static_cast, ולא בתחביר השני, המיועד בעיקר כדי לתמוך בקוד C ישן.

המרת down-casting עשויה להיות לא חוקית. ניתן לבצע המרת down-casting בטוחה על ידי dynamic_cast. לדוגמא:

```
Manager &mRef = dynamic_cast<Manager &>eRef;
```

במידה ואי יהיה אפשר לבצע את ההמרה, שפת C++ תזרוק חריגה מסוג bad_cast.

נביט בדוגמא לתוכנית בדיקה האם אובייקט הוא מסוג מסוים:

```
#include <iostream>
#include <cstdlib>
using namespace std;

class A
{
public:
    virtual void f() {};
};
```

```
class B : public A
{
public:
    void f() {};
};
class C
{
public:
    void f() {};
};


int main()
{
      A *a1 = new A;
      A *b1 = new B;
      C *c1 = new C;

      if (B* ref1 = dynamic_cast<B*>(b1))
            cout << "b1 is B" << endl;
      else
            cout << "b1 isn't B" << endl;

      if (C* ref1 = dynamic_cast<C*>(b1))
            cout << "b1 is C" << endl;
      else
            cout << "b1 isn't C" << endl;

      return 0;
}
```

פלט התוכנית:

```
b1 is B
b1 isn't C
```

- נשים לב שבתנאי ה-if היה = בודד. זו איננה שגיאה, אלא התחביר. אנחנו בודקים האם ref1 למשל הוא NULL או לא למעשה.

4.3.4. סמנטיקת ערכים, Coercion, Polymorphism

פולימורפיזם פועל על קוד, על משתנים אך לא על ערכים. Coercion לעומת זאת פועל להמיר ערך מסוג אחד לערך מסוג אחר, לרוב על ידי איבוד של חלק מהתוכן של האובייקט המקורי.

הורשה בשפת C++ מגדירה coercion בין המחלקה הנורשת אל מחלקת הבסיס. ה-coercion הוא Parametric – מוגדר אוטומטית לכל תתי הסוגים. ה-coercion נעשה על ידי שליפת תת אובייקט (sub object) מתוך האובייקט המקורי.

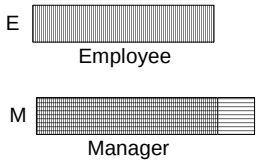
מהו תת אובייקט?

```
class Base
{
      // ...
};

class Derived: public Base
{
      // ...
};
```

לכל אובייקט של המחלקה Derived יש תת אובייקט שהוא מהמחלקה Base. במקרים רבים נרצה להתייחס אל תת-אובייקט זה.

אם Manager יורש מ-Employee, אז יש לו חלק משותף ל-Employee.




```
Employee E;
Manager M;

E = M;      // OK - valid coercion
M = E;            // Error - No coercion is defined
```

ניתן להמיר מסוג האב לסוג הבן על ידי קיצוץ השדות הנוספים. הכיוון השני של המרה אינו מוגדר.

5. Overriding

5.1. Overriding ו-Non strict Inheritance

5.1.1. דוגמא – הורשה ו-Overriding

נציג כעת סוג מורחב יותר של הורשה – Non strict inheritance.

Strict inheritance איפשר להרחיב את המחלקה המקורית על ידי הוספת שדות ופונקציות בלבד. Non strict inheritance ייתן לנו כוח נוסף.

הקוד הבא ממש מחלקה המייצגת מערך. המחלקה אינה בודקת אם המשתמש חורג מגבולות המערך.

```
class Array
{
public:

	//
	// Ctor, Dtor
	//
	explicit Array(int len_) :
		len(len_), buff(new int[len]) {}
	~Array(void) { delete[] buff; }

	//
	// Public Methods and operators
	//
	int size(void) const { return len; }
	int& operator[](int i)
	{
		return buff[i];
	}
	int operator[](int i) const
	{
		 return buff[i];
	}

private:
	const int len;
	int* const buff;
	Array(const Array&);
	Array& operator=(const Array&);
};
```

<u>מעט הסברים על הקוד</u>

השימוש במילה explicit: בנאי (constructor) המקבל פרמטר אחד משמש בשפת C++ כאופרטור המרה. אם הבנאי מוגדר כ-explicit אז הוא אינו משמש כאופרטור המרה להמרה סמויה.

נניח שקיימת פונקציה עם החתימה:

```
void f(Array arr);
```

המצב הבא הוא תקין. הפונקציה מקבלת ערך מסוג Array:

```
Array arr(10);
...
f(arr);
```

המצב אותו אנו רוצים למנוע הוא מקרה שמשתמש יקרא בטעות לפונקציה עם int, לדוגמא:

```
f(3);
```

אם המשתמש באמת רוצה לקרוא לפונקציה עם מערך חדש בגודל 3, אז במידה ואנחנו משתמשים במילה explicit הוא יצטרך לכתוב זאת כך:

```
f(Array(3));
```

Copy Ctor ואופרטור= הממומשים כ-private

בונה המחלקה לא רצה לאפשר למשתמשים במחלקה להשתמש בתכונות אלו, אולם הוא רצה לכתוב אותם כדי לאפשר לעצמו מימוש יעיל יותר של הקוד הפנימי של המחלקה. לפיכך מתודות אלו מומשו כמתודות פרטיות.

נרצה כעת ליצור מחלקה חדשה, שתהיה מערך לו קיימת בדיקת גבולות.

על מנת לעשות שימוש חוזר בקוד, נירש מהמחלקה Array. בעמוד הבא נציג את קוד המחלקה החדשה.

```
class CheckedArray: public Array
{
      class RangeError {};
public:
      explicit CheckedArray(int len) : Array(len) {}
      int& operator[](int) throw (RangeError);
};

int& CheckedArray :: operator[](int i)
      throw (CheckedArray :: RangeError)
{
      if (0 > i || i >= size()) throw RangeError();
      return Array :: operator[] (i);
}
```

ההורשה בדוגמא הינה הורשת public, כלומר – כל מה שהיה public במחלקה המקורית, הוא גם ה-public של המחלקה החדשה. כל מה שהיה protected במחלקה המקורית, הוא גם protected במחלקה החדשה.

נשים לב שבהורשה ה-constructor אינו נורש אל המחלקה החדשה. המחלקה החדשה חייבת להגדיר constructor משלה שיקרא ל-constructor המתאים של Array, וזאת מכיוון של-Array אין בנאי חסר פרמטרים.

גם בנאי המחלקה החדשה מוגדר כ-explicit, מאותם נימוקים שבכללם Array היה מוגדר כזה.

Overriding .5.1.2

המחלקה מבצעת **חפיפה (Overriding)** של האופרטור []. האופרטור החדש מחליף את זה של המחלקה הישנה. הגרסה החדשה מוסיפה את מגבלת התחום הרצויה. במידה והמשתמש חרג מתחום המערך – המחלקה זורקת exception.

5.1.3. Non strict inheritance

Non strict inheritance: ידוע גם בשם הכולל **הורשה**.
זהו מנגנון חזק יותר מהמנגנון של Strict Inheritance. מנגנון זה מאפשר לנו:

- הוספת פעולות חדשות לאובייקטים מהמחלקה החדשה.
- הוספת שדות חדשים לאובייקטים מהמחלקה החדשה.
- מימוש שונה / הרחבת הודעות שהיו קיימות במחלקה המקורית.

התוצאה: מנגנון חזק יותר באמצעותו ניתן לבצע דברים חדשים.

נשים לב שלא ניתן, גם בעזרת מנגנון זה, להגדיר מחדש שדות, אלא רק התנהגות.
לא ניתן להגדיר בבן משתנה שהוא char x אם במחלקת האב קיים שדה int x.
מה היינו עושים אם היינו רוצים להגדיר שדה שכן ישתנה בבנים? הפתרון האלגנטי ביותר הוא לדאוג שהשדה יהיה מסוג מצביע לאובייקט, ולא מסוג ערך. במקרה זה הבן יוכל לשים באותו מצביע משתנה מסוג אחר שנורש מהסוג המקורי שהיה בו.

5.1.4. סוגי Overriding

- **החלפה (Replacement)** – המימוש החדש של הפונקציה מחליף את המימוש שהיה קיים במחלקת הבסיס – לבן יש מימוש חדש של הפונקציה והוא מפסיק להכיר את הפונקציה שהייתה קיימת במחלקת האב. בשפת Eiffel, למשל, קיים סוג Override זה.
- **עידון (Refinement)** – הפונקציה החדשה "מעדנת" את הקוד של הפונקציה הראשונה – היא מסוגלת להשתמש בפונקציה המקורית המוחלפת. במקרים רבים הפונקציה הנורשת קוראת למתודה המקורית לביצוע הפעולה (שימוש למשל ב-super בשפת Little Smalltalk).

כאשר אנחנו מדברים על עידון, מתעוררת שאלה חדשה: מה הקשר בין הפונקציה f שבמחלקת האב לבין הפונקציה f שנכתבה מחדש במחלקת הבן.
השאלה שנשאל היא על מי מוטלת האחריות ליצור ולהשתמש בקשר.

הגישה המקובלת – **Alpha Refinement** – המחלקה החדשה אחראית על הקשר עם הגרסא הישנה.
גישה נוספת הינה **Beta Refinement** – מחלקת האב אחראית לקרוא לגרסת הפונקציה של המחלקה הנורשת על ידי קוד מיוחד במחלקת האב. אם לא קיים קוד כזה – גרסה חדשה של הפונקציה שתיכתב במחלקת הבן פשוט לא תופעל.

תזכורת: אנו רוצים לדאוג לשמירה על תאימות: אם פונקציה f המקבלת אובייקט ממחלקת הבסיס עושה משהו ובתנאים מסוימים ניתן לקרוא לה, צריך להיות ניתן לקרוא לה באותם תנאים גם עבור אובייקט של כל מחלקה נגזרת ולקבל תוצאה נכונה.

שימוש ב-Beta Refinement מאפשר לנו לשלוט על התאימות בצורה טובה יותר. מחלקת האב יכולה להגדיר מה בדיוק יהיה מותר למחלקות הנגזרות לשנות ומה לא.
חסרון בגישה זו הוא שמחלקת האב צריכה להכיר את הצרכים שיהיו למחלקת הבנים. אם יתעורר צורך עליו לא חשבנו, ייתכן ויהיה צורך לערוך את גם מחלקת האב ולא רק את מחלקת הבן על מנת לספק את הצורך החדש.

Dynamic Binding .5.2

5.2.1. הקדמה – הרעיון ושימושים

Binding – קישור בין ההודעות למתודות.
כאשר יש בשפה גם overriding וגם inclusion polymorphism מתעוררת השאלה מתי מתבצע הקישור הנ"ל.
במקרה כזה אותו קטע קוד יכול להתייחס לאובייקטים שונים ממחלקות שונות, שלכל אחד מהם מימוש משלו של ההודעה.
קישור יכול להתבצע בזמן ההידור, בזמן הקישור או בזמן הריצה. בזמן הריצה – ייתכן שנוכל לבצע את ה-binding עם תחילת התוכנית, ויתכן שרק כשנגיע ממש לביטוי, נדע מה המתודה המתאימה לה צריך לקרוא.

Early Binding (static) זהו קישור בזמן הידור. המהדר משתמש בסוג המשתנים כדי לבצע את הקישור בזמן ההידור / ה-linking.
Late Binding (dynamic) זהו מצב בו הקישור בין ההודעה למתודה נעשה בזמן הריצה.

Static binding הוא יעיל יותר, אך גמיש פחות. בדיקות סטאטיות מובילות לקוד בטוח יותר.
Dynamic binding הוא איטי יותר, מכיוון שאנחנו מבצעים פעולות נוספות בזמן ריצה. הוא דורש מנגנון של dynamic type checking, אך עם זאת הוא נותן לנו גמישות וכוח גדולים יותר.
למרות חסרונותיו, הכוח ש-dynamic binding נותן לנו גורם לתכנות מונחה עצמים להסתמך עליו באופן חזק.
דוגמא לשימוש: ניתן להגדיר מחלקה שהיא חיה כללית, וכן מחלקות נורשות המייצגות חיות שונות. לאחר מכן מכולה יכולה להכיל מצביעים מסוג חיות, ולנהל את החיות השונות, להפעיל את הפונקציות המיוחדות להן.

מתי מתבצע החיפוש מהי מחלקת הבסיס של אובייקט? במקרה של Smalltalk ו-Pascal למשל, מתקיים כי super ו-inherited ניתנים לחישוב סטאטי – בזמן קומפילציה.
ככלל המהדרים ינסו כמה שניתן לבצע קישורים בזמן הידור, וישאירו את מה שלא ניתן לדעת בזמן קומפילציה אל זמן הריצה.

5.2.2. Downcasting מול Dynamic Binding

ניתן בעזרת downcasting לכאורה לוותר על הצורך ב-dynamic binding, לדוגמא:

```
void draw(Shape *p)
{
      if (Circle *q = dynamic_cast<Circle *>p) {
            // Draw circle
            ...
      } else if (Line *q = dynamic_cast<Line *>p) {
            // Draw line
            ...
      } else if (Rectangle *q = dynamic_cast<Rectangle *>p) {
            // Draw rectangle
            ...
      }
            ...
}
```

למרות ששיטה זו תעבוד, היא לא מומלצת. הסיבה: אם נשתמש בה, אז אם נוסיף מחלקה חדשה אל מודל המחלקות שלנו, נצטרך ללכת לכל המקומות בקוד שיש בהם if כזה, ולשנות אותם. לעומת זאת אם היינו משתמשים ב-dynamic binding לא היה צורך.

5.2.3. High Level Method

בעזרת Dynamic Binding ניתן ליצור מתודות פולימורפיות – מתודות שיבצעו דברים שונים לגמרי עבור אובייקטים שונים.

```
void Shape::move(Point delta)
{
      hide();
      location += delta;
      draw();
}
```

hide() ו-draw() הן מתודות וירטואליות.

נבדיל בין מספר סוגי מתודות:

- **Low Level Methods** – ממומשים רק באמצעות data members של המחלקה – לא מושפעים מ-overriding.
- **High Level Methods** – ממומשים גם בעזרת מתודות אחרות העשויות להיות פולימורפיות.
- **Outer Level Methods** – ממומשות רק על ידי פונקציות חיצוניות – פולימורפיות לגמרי.

5.2.4. Static Typing-ו Dynamic Binding

Static typing מבטיח שהמתודה שאנחנו רוצים לקרוא לה אכן קיימת באובייקט שאליו אנחנו פונים.

Dynamic Binding דואג שבכל מקרה המתודה המתאימה לאובייקט תיקרא.

שילוב של static typing וגם dynamic binding נותן לנו כוח בשילוב עם בטיחות. רוב השפות הן כאלה: Eiffel, C++, Java, Turbo-Pascal ועוד.

5.2.5. מימוש Dynamic Binding

כיצד ממומש overriding? על ידי מצביעים לפונקציות.
לכל אובייקט יש טבלה של מצביעים אל המימוש המתאים של הפונקציות הוירטואליות שלו.
טבלה זו – Virtual Methods Table, מכונה בקיצור **vtbl**.
נגדיש: רק פונקציות וירטואליות מופיעות ב-vtbl. פונקציות שאינן וירטואליות מקושרות סטאטית בזמן ההידור. כמו כן, פונקציה שהיא overloading, מספר פונקציות עם מספר פרמטרים משתנה, מופיעים כמספר כניסות נפרדות בטבלה.

טבלת המצביעים לפונק' מתעדכנת כשנוצר עצם מהמחלקה.
כידוע כאשר יוצרים אובייקט ראשית נקראת הפונקציה הבונה של האב הקדום ביותר, אחרי זה של הצאצא שלו וכך הלאה. הרעיון הוא שבהתחלה, כל הפונקציות בטבלת המצביעים מצביעות אל הפונקציות המתאימות של האב. במהלך בניית האובייקט, נניח כי הגענו אל בנאי של מחלקה המציעה מימוש שונה לאחת המתודות, אזי המתודה מעדכנת את המצביע אל הפונקציה המתאימה.

נדגיש: בזמן ריצת התוכנית, ה-vtbl נמצאת בזיכרון ומאפיינת את האובייקטים של המחלקה.
במצב אידיאלי, היינו רוצים שלכל מחלקה תהיה vtbl יחידה, כך שנוכל לזהות שכל העצמים המצביעים אל אותה vtbl שייכים למחלקה הנ"ל.
ב-C++ מודל הקומפילציה הוא כזה שאינו מאפשר לנו להשיג מטרה זו. המודל הוא מודל קומפילציה נפרדת. כל קובץ עובר קומפילציה בנפרד. אפשר לאסוף לתוכנית אחת קבצים שקומפלו בזמן שונה ואפילו במחשבים שונים. לפיכך - המהדר צריך ליצור vtbl עבור כל קובץ בפרויקט.

כאמור לכל אובייקט יש טבלת מצביעים לפונקציות המתאימה לו. הטבלה ממוקמת במקום כלשהו בזיכרון. על מנת לגשת אליה, קיים בכל אובייקט שדה נוסף בשם vptr, המצביע אל הטבלה הנ"ל. מתעוררת השאלה היכן נמקם את מצביע זה.

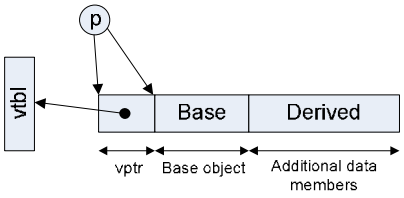
הגישה של שפת C++ מכונה "**גישת הסופרמרקט**". גישה זאת אומרת את הדבר הבא: אנחנו לא הולכים לשלם – לא במקום ולא בזמן ריצה, עבור אפשרויות של השפה שאנחנו לא משתמשים בהם – "לא השתמשת – לא שילמת". הדבר נכון גם לגבי dynamic binding. אם מתכנת לא משתמש ב-dynamic binding, על השפה לדאוג שהוא לא ישלם עלויות עבור תמיכה במנגנון זה.
מיד נראה שגישה זו יוצרת מספר סיבוכים במימוש מנגנון ה-dynamic binding.

Borland .5.2.5.1

Borland C++ בחרו למקם את ה-vptr בתור השדה הראשון באובייקט.
גישה זו הינה בעייתית מול הגישה של שפת C++. מכיוון ש-vptr נמצא רק בעצמים של מחלקות שיש בהם dynamic binding, קיימת בעיה במקרים בהם מדובר במחלקה שאינה משתמשת ב- dynamic binding. הפתרון של Borland – במקרים אלו ערכו של ה-vptr הינו NULL.

חסרון בגישה זו: גם כאשר איננו משתמשים ב-dynamic binding – אנחנו משלמים במקום.

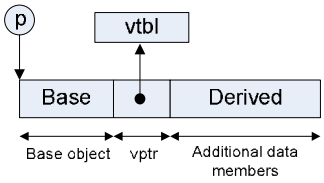
בעיה נוספת: כתובת הבסיס של האובייקט משתנה כאשר אנחנו מבצעים Upcasting.
הכתובת של p, התחלת האובייקט, משתנה כאשר מתבצע upcasting מ-Derived אל Base.



מכאן: upcasting לפי Borland – אומר שכתובת האובייקט משתנה. זה אומר שהמרות בשפת C++ הרבה יותר מסובכות מאשר בשפת C (שמה פשוט נתנו משמעות חדשה לכתובת הישנה).
נשים לב ש-casting בסגנון C לא היה עובד כאן כראוי – והיה מביא לתוצאות שגויות.
החשיבות של ההמרות: בכל פעם שעושים downcasting וקוראים לפונקציה של הבסיס, צריך לחשב את this – פגיעה בביצועי התוכנית.

GNU .5.2.5.2

ה-vptr ממוקם תמיד לפני המחלקה הראשונה שלה יש פונקציה וירטואלית.
במקרה זה המימוש של casting הוא פשוט ביותר, אולם המימוש של הרצת פונקציה וירטואלית מסובך יותר. סביבת הריצה צריכה אלגוריתם שימצא היכן נמצא ה-vptr.



G++ .5.2.5.3

g++ בחרה באסטרטגיה שלישית. לשים את ה-vtbl ptr אחרי האובייקט.
כלומר: מובטח כמעט ש-vptr לא מופיע בתחילת העצם (יוצא דופן – במחלקת הבסיס אין שדות נתונים).

Dynamic Binding ויעילות .5.2.5.4

ברור כי dynamic binding פוגע ביעילות. השאלה היא עד כמה.
בשפות שהן statically typed הפגיעה איננה חמורה – על כל גישה לפונקציה וירטואלית אנחנו משלמים ב-2-3 גישות למצביעים כדי לאתר את מיקומה – מספר קטן של מחזורי שעון.
בשפות כגון Smalltalk העונש גדול יותר – גרף הירושה יכול להשתנות בזמן ריצה, ולכן יש צורך ביותר בדיקות.
ב-Java, C# וכו' בניגוד ל-C++, ניתן לראות את כל הקוד בזמן קומפילציה (השפות הינן non-separated compilation), ולכן הן מסוגלות לעשות אופטימיזציה כתוצאה ממידע נוסף על המערכת ועל היררכיית הירושה). בצורה כזו הן מסוגלות לצמצם את הפגיעה בביצועים הנגרמת מ- dynamic binding. ב-C++ עקרונית ניתן לבצע אופטימיזציה דומה שתמומש ב-linker, אולם אופטימיזציה כזו איננה מבוצעת כיום.

Inline Virtual Function .5.2.5.5

הפגיעה האמיתית בביצועים כאשר אנחנו משתמשים ב-dynamic binding נעשית לא בקריאה לפונקציות עצמן, אלא בעובדה שאנחנו מפספסים בגללן הזדמנויות בהן היינו יכול להשתמש ב-virtual.

מתי נוכל להשתמש ב-inline בשילוב עם virtual functions?

- במקרה הבא לא נוכל להשתמש ב-inline – הפונקציה איננה ידועה בזמן ההידור.

```
void SomeClass :: some_function_member(void)
{
      some_virtual_function_member();
}
```

- אם אנחנו מקבלים אובייקט (ולא מצביע או משתנה התייחסות לאובייקט), אנחנו יכולים להגדיר את הפונקציה המקבלת כ-inline:

```
void f(SomeClass a)
{
      a.some_virtual_function_member();
}
```

- אם קוראים לפונקציה וירטואלית בצורה מפורשת (כך שברור איזה פונקציה תיקרא באופן מוחלט) אזי גם ניתן להשתמש ב-inline:

```
void f(SomeClass& a)
{
      a.SomeClass::some_virtual_function_member();
}
```

- בתוך ctor ניתן לעשות inline לפונקציה וירטואלית בשפת C++. ב-C++, כאשר ה-ctor של מחלקה נגזרת מתחיל לעבוד הוא מתחיל ב-ctor של ההורים – הבניה בעצם מתחילה מה-ctor של מחלקת הבסיס, ולאחר מכן לפי עץ ההורשה של המחלקות בדרך. הנקודה החשובה היא שגם כשבונים מחלקה C שנורשת מ-B שיורשת מ-A, בתחילת הבניה המחלקה היא A. במהלך הבניה, vptr מתעדכן כל פעם ביחס למחלקה שה-ctor שייך לה. לפיכך קריאה לפונקציה וירטואלית בתוך ה-ctor אינה ממש dynamic-binding ולכן הקומפיילר יכול לדעת לאיזה מתודה מתכוונים בכל קריאה.

בשפת Java ניסו לפשט את המנגנון. הגישה ב-Java היא שראשית יוצרים את העצם כולו עם כל השדות שבו ומאתחלים אותם להיות 0. לאחר מכן מתחילים לבנות את האובייקט מהבסיס בדומה לשפת C++ ומעדכנים את השדות לפי הבנאים שבשרשרת ההורשה. כעת ניתן לקרוא למתודות לפי ה-dynamic binding. הרעיון מאחורי גישה זו – הטיפוס הסופי של האובייקט נקבע עם יצירתו.

5.3. תאימות – conformance

5.3.1. הצגת הנושא

שמורות של המחלקה – תכונות בסיסיות המשותפות לכל איברי המחלקה.
פונקציה המשנה פונקציה אחרת (overriding) צריכה לשמור לפחות על השמורות הישנות. בכל מקום בו היינו מסוגלים להשתמש באובייקטים מסוג מחלקת הבסיס, אנחנו צריכים להיות מסוגלים להשתמש באובייקטים ממחלקת הבן.
למחלקת הבן מותר להגביל יותר את השמורות, אך לא להגביל פחות. למשל, אם ערך של שדה מסוים במחלקה האב נקבע שהוא בין 1 ל-100, מחלקת הבן יכולה להחליט שתחום ערכים חוקי בשבילו הוא בין 1 ל-50, אולם היא איננה יכולה לפתע להחליט שהערכים בו יהיו בין 1 ל-200 – זו תהיה פגיעה בתאימות – ייתכן שיהיה מצב בו נוכל להשתמש באובייקט של מחלקת האב, אך לא באובייקט של מחלקת הבן.

המצב הכללי: על עצם a מפעילים מתודה f שמקבלת עצם b ומחזירה תוצאה אל c:

```
c = a.f(b);
```

תאימות: אם השורה הנ"ל עבדה נכון לפני ההורשה, היא תעבור גם אחרי.
הפונקציה f() החדשה צריכה לעמוד לספק לפחות את אותם שירותים שהפונקציה הקודמת נתנה.
מכאן – הערך של הפרמטר המתקבל הינו לפחות בתחום הערכים שהפונקציה המקורית קיבלה, אם כי יכול להגדיל אותו. הערך שהפונקציה מחזירה, חייב להיות תת תחום של קבוצת הערכים שהפונקציה הישנה החזירה.

<u>דגש נוסף</u>: אם קוד מסוים עבר הידור לפני שהגדרנו הורשה, הוא צריך להיות בר-קומפילציה גם אחרי ההורשה. כמו כן, בהינתן מחלקת D הנורשת מ-A והאובייקטים a, d שהינם עצמים השייכים למחלקות אלו, הקוד הבא חייב אף הוא לעבור קומפילציה:

```
c = d.f(b);
```

וזאת מכיוון שכל מה שעבד עבור אובייקטים של המחלקה A, חייב לעבוד גם עבור אובייקטים של D.

Conformance and Overriding .5.3.2

אם נסכם, נאמר כי הודות ל-dynamic binding וכן לתכונת הפולימורפיזם, המשתמש לא יכול לדעת מראש איזו גירסה של מתודה וירטואלית תקרא בפועל כאשר הוא משתמש בה. כתוצאה מכך מתעורר הצורך בתאימות. התאימות צריכה להתבטא במספר מישורים:

- רמת הגישה של הפונקציה.
- החוזה עם הפונקציה - התנאים המקדימים לקריאה, התנאים הסופיים והערכים שיכולים להיזרק.
- חתימת הפונקציה - הפרמטרים של הפונקציה והערך המוחזר שלה.

"Same or better" Principle - פונקציה חופפת חייבת לספק לפחות את אותה פונקציונליות כמו הפונקציה הנחפפת. לפונקציה החופפת אסור להפתיע את המשתמש.

רמת הגישה של הפונקציה - Access Conformance .5.3.2.1

Access Conformance: כל הגרסאות של הפונקציה צריכות להיות בעלות אותה רמת נגישות (protected/public).

שפת Smalltalk שומרת על עיקרון זה. שפת C++ לעומת זאת משאירה זאת באחריות המתכנת.

נביט בדוגמא הבאה:

```
class Base
{
public:
      virtual void f(void);
};

class Derived : public Base
{
private:
      void f(void);
} y, *py = &y;

Base *px = py;

py->f();  // Error! Derived::f is private.
px->f();  // OK! (but breaks encapsulation.)
```

בדוגמא זו על ידי מצביע ממחלקת הבסיס הצלחנו להריץ את הפונקציה f() של המחלקה הנגזרת, שהיא מוגדרת כ-private.

דרישת התאימות של רמת הגישה דורשת: הפונקציה החדשה, הדורסת, חייבת להיות נגישה לכל אלמנט שהפונקציה המקורית היתה נגישה לו.
אם מקטינים את זכויות הפונקציה, פגענו בתאימות. אם מגדילים את הזכויות (מ-protected ל-public למשל) לא פגענו בתאימות.

5.3.2.2. Contract Conformance – החוזה עם הפונקציה

תנאים מקדימים: הפונקציה החופפת יכולה לדרוש את הדרישות שהפונקציה המקורית דרשה, או פחות.
תנאי סיום: הפונקציה החופפת חייבת להבטיח לפחות את מה שהפונקציה המקורית הבטיחה.
חריגות: לפונקציה החופפת אסור לזרוק חריגות שהפונקציה המקורית לא זרקה.

הרעיון מאחורי דרישות אלו: "Same or better principle".

5.3.2.3. Signature Conformance - חתימת הפונקציה

האלמנטים המרכיבים את חתימת הפונקציה:

- Input Parameters.
- Output Parameters.
- Input-Output Parameters.
- הערך המוחזר של הפונקציה.

נגדיר מספר דרכים בהן הפונקציה החופפת יכולה לשנות את הגדרת הפונקציה:

No-variance: הסוג בחתימה אינו משתנה.
Co-variance: השינוי בסוג החתימה הוא באותו כיוון כמו ההורשה – במורד עץ ההורשה.
Contra-variance: השינוי בסוג החתימה הוא בכיוון הפוך להורשה – במעלה עץ ההורשה.

תאימות דורשת:

- Contra-variance של ה-input parameters.
- Co-variance של ה-output parameters.
- No-variance של ה-input-output parameters.
- Co-variance של הערך המוחזר.

6. ירושה מרובה – סקירה קצרה

מסמך זה אינו נכנס לעומק לנושא הירושה המרובה, אולם רציתי להזכיר את הרעיון הכללי – מהי ירושה מרובה.

בגדול – ירושה מרובה היא ירושה בה מחלקה נורשת ממספר מחלקות, ולא ממחלקה בודדת.
ירושה מרובה מבוצעת בשפת C++ בדומה להורשה יחידה. לאחר שם המחלקה, וכתיבת נקודותיים, מציינים רשימה של מחלקות מהן נורש האובייקט. לפני כל שם מחלקה יצוין סוג ההורשה. סדר המחלקות ברשימה משפיע על מבנה האובייקט ועל סדר הפעלת הבנאים וההורסים.
מבחינה לוגית טיפוס המחלקה הנורשת מכל המחלקות מוכל בחיתוך של כל הטיפוסים מהם הוא יורש.

מתי נשתמש בירושה מרובה? טענה האהובה על אנשי Java היא שאין מקרים בהם חייבים להשתמש בהורשה מרובה. מצד אחד ניתן לתכנת ללא הורשה מרובה, אולם מצד שני יש מקרים רבים בהם הורשה מרובה היא הפתרון הטבעי לבעיה. הורשה מרובה זכתה לשם הרע שלה מכיוון שהיא יוצרת מבנים שלפעמים קשה להבין, ויש מספר בעיות הייחודיות לה. אם זאת, ניתן לומר דברים אלו על כל נושא בתכנות – כל נושא שלא שולטים בו טוב, יתכן ויהיו איתו בעיות, ולכן אין לפסול הורשה מרובה על הסף, כמו שרבים עושים. יתרה מזאת, ישנם מקרים בהם חייבים להשתמש בהורשה מרובה, לדוגמא כאשר יש לנו צורך להשתמש בשירותים הניתנים תחת protected של מספר מחלקות שונות.

בעיות בירושה מרובה:

1. **כפל משמעות**: פונקציה בעלת אותו שם המוגדרת ביותר מבסיס אחד.
2. **Diamond** – מחלקה אחת הקיימת יותר מפעם אחת בעץ הירושה. בשפת C++, בגלל גישת ההידור המחולק, כל מחלקה מחליטה אם לשתף לרמות הבאות את המחלקה/ות מהן היא יורשת. אם אחת המחלקות לא הסכימה לשתף מחלקות אלו, אז העותק ממנו היא יורשת יהיה נפרד. כדי לשתף את מחלקת הבסיס יורשים virtual בנוסף לסוג הירושה המבוקש (public virtual).

- כפל משמעות: תתכן בעיה כזו עבור שדה נתונים או שיטה באותו שם: במצב של שכפול מחלקת הבסיס יש שתי אפשרויות לבצע up-casing. בשפת C++, כפל משמעות של שיטה\משתנה יגרום לשגיאות קומפילציה.

7. מקורות

מקורות באנגלית:

1. Abadi, M. & Cardelli, L.(1996), "**A Theory of Objects**"
2. Dr. Yossi Gil – "**Object Oriented Programming – Lectures Notes**"
3. Luca Cardelli and Peter Wegner (1985), "**On Understanding Types, Data Abstraction, and Polymorphism**"
4. Scott Danforth and Chris Tomlinson (1988), "**Type theories and object-oriented programming**"
5. Uppsala University (1998), "**Software Design**"
6. Uppsala University (1998), "**Polymorphism**"
7. William R. Cook (1994), "**Inheritance is not subtyping**"
8. Wm. Paul Rogers (2004), "**Reveal the magic behind subtype polymorphism**"
9. Zhanshan Gao (1999), "**Inheritance in Object-Oriented Theories**"

מקורות בעברית:

1. אדר נ. (2004), "**שפות תיכנות**"
2. כהן א. (2004), "**סיכומי הרצאות בתכנות מונחה עצמים של ד"ר יוסי גיל**"
3. אדר נ. (2005), "**סיכומי הרצאות בתכנות מונחה עצמים של ד"ר יחיאל קימחי**"