



# עיצוב על פי חוזה (Design by Contract)

אוהד ברזילי

אוניברסיטת תל אביב



המצגת מכילה קטעים מתוך מצגת של פרופ' עמירם יהודאי  
ע"פ הספר:

Object-Oriented Software Construction, 2nd edition,  
by Bertrand Meyer (Prentice Hall) .

כל הזכויות שמורות למחברים



# בנייה שיטתית של תוכנה

- מימוש מחלקות מצריך התייחסות לשיקולי יציבות ולנכונות
- איך מבצעים את המעבר בין ADT לבין מימוש המחלקה בשפת התכנות?
- איך מתמודדים עם שגיאות המתגלות בזמן ריצה?
- השיטה המוצגת כאן נקראת "עיצוב ע"פ חוזה" (DbC)



# נכונות

- נכונות מוגדרת יחסית למפרט (specification) נתון
- בצורה מתמטית:  $\{P\} A \{Q\}$ 
  - התוכנית  $A$  רצה במצב מסוים, שבו מתקיים  $P$ , ומסתיימת במצב שבו מתקיים  $Q$
  - תוכנית כמתמיר מצבים
- תנאים חלשים:
  - $\{false\} A \{...\}$
  - $\{...\} A \{true\}$
- שני התנאים לעיל חלשים (טריויאליים) מכיוון שכל תוכנית  $A$  מקיימת אותם (רק השני מבטיח סיום)



# P ו-Q כביטויים בולאנים

- ניתן להכליל את P ו-Q בגוף התוכנית (כהערות)
  - עליהם להיות ביטויים בולאנים שערכם TRUE
  - גם אם אין תמיכה בשפת התכנות (או ע"י כלים חיצוניים) לשערוך P ו-Q יש להם חשיבות קונספטואלית גבוהה לנכונות התוכנה ויציבותה
  - ניתן לספק עבור תנאים שמות לוגיים:
    - Positive:  $m > 0$
    - Not\_Void:  $p \neq \text{null}$



# תנאי קדם ותנאי בתר

## precondition and postcondition

- עבור כל פונקציה נוכל להגדיר ביטויים בולאנים משני סוגים:
  - תנאי קדם: ביטויים שעליהם להיות נכונים לפני ביצוע הפונקציה
  - תנאי בתר: ביטויים שעליהם להיות נכונים בסיום ביצוע הפונקציה
- ביטוי  $\$prev(exp)$  המתאר את ערכו של  $exp$  לפני תחילת ביצוע הפונקציה
- הביטוי  $\$result$  המתאר את הערך המוחזר של הפונקציה



# מחסנית ע"פ חוזה

```
template<class T>
class STACK1
{
public:

    /** Is stack empty? */
    bool empty() const {...} // Same as count==0

    /** Top element
     * @pre: !empty()
     */
    T top() const {...}
```

# מחסנית ע"פ חוזה

```
/** Is stack full? */
bool full() const {...}

/** Add x on top
 * @pre: !full()
 * @post: !empty()
 * @post: top() == x
 * @post: count == $prev(count) + 1
 */
void push(T x) {...}
```







# מחסנית ע"פ חוזה

```
/** Remove top element
 * @pre: !empty()
 * @post: !full()
 * @post: count == $prev(count) - 1
 */
void pop() {...}
```

```
private:
```

```
/** Number of stack elements */
int count;
};
```

# החוזה



- תנאי הקדם מחייב את הלקוח (המחלקה הקוראת לפונקציה)
- תנאי הבתר מחייב את הספק (המחלקה המממשת את הפונקציה)
- הספק: "אם את מתחייבת לקרוא לי רק כאשר @pre מתקיים, אז אני מתחייב להביא את התוכנית למצב שבו בסוף ביצוע הפונקציה @post מתקיים"
- אם תנאי הקדם לא מתקיים הספק רשאי לעשות כרצונו

# DbC in Eiffel

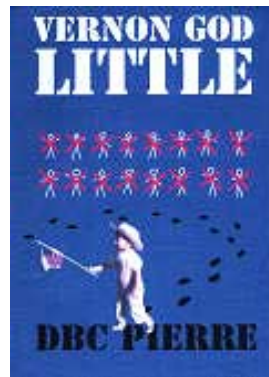
○ בשפת Eiffel:

● תנאי הקדם מסומן במילה השמורה **require**

● תנאי הבתר במילה **ensure**

● ביטוי ה- `$prev` מסומן ע"י המילה השמורה

**old**

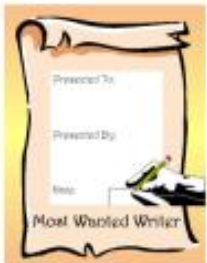


# דוגמא

רווח	התחייבות	push
יודע שהפעולה בוצעה	קורא רק כשהמחסנית אינה מלאה	לקוח
פשטות במימוש	העצם החדש יוכנס לראש המחסנית	ספק

# ניסוח החוזה

- הימנעי מבדיקות כפולות (תכנות מתגונן – defensive programming). גוף של פונקציה לעולם לא יבדוק את התנאי המקדים של הפונקציה
- אין לראות את החוזה כמנגנון לבדיקת תקינות הארגומנטים (אם כי הוא עשוי להכיל בדיקות כאלה)
- על החוזה אסור לשמש לשינוי מהלך התוכנית או לשינוי מצב התוכנית (side effect)
- השתמשי הפונקציות פשוטות
- אחריות ברורה



# הפרת החוזה

- כאשר במהלך ריצת התוכנית אחד הפרדיקטים הוא false אז:
  - תנאי מקדים – באג אצל הלקוח
  - תנאי בתר – באג אצל הספק





# STACK2

```
template<class T>
class STACK2 {

public:
    /** Allocate stack for maximum of n elements
     * @pre: positive_capacity: n >= 0
     * @post: capacity_set: capacity == n
     * @post: array_allocated: representation.capacity >= n
     * @post: stack_empty: empty()
     */
    STACK2(int n) :    capacity(n), count(0),
                   representation(capacity)
    {}
}
```



# STACK2

```
/** The maximum number of stack elements
 */
const int capacity;

/** Top element
 * @pre: not_empty: !empty()
 */
T top() const
{
    return representation[count-1];
}
```





# STACK2

```
/** Is stack empty?  
 * @post: empty_definition: $result == (count == 0)  
 */  
bool empty() const  
{  
    return count == 0;  
}  
  
/** Is stack full?  
 * @post: full_definition: $result == (count == capacity)  
 */  
bool full() const  
{  
    return count == capacity;  
}
```



# STACK2

```
/** Add x on top
 * @pre: not_full: !full()
 * @post: not_empty: !empty()
 * @post: added_to_top: top() == x
 * @post: one_more_item: count == $prev(count)+1
 * @post: in_top_array_entry: representation[count-1] == x
 */
void push(T x)
{
    representation.insert(
        representation.begin()+ count , x);
    count++;
}
```



# STACK2

```
/**
 * @pre: not_empty: !empty()
 * @post: not_full: !full()
 * @post: one_fewer: count == $prev(count)-1
 */
void pop() { count--; }
```

private:

```
/** The array used to hold elements */
vector<T> representation;

/** Number of stack elements */
int count;
};
```



# עוד על ניסוח החוזה

- תנאי בתר החוזר על גוף הפונקציה אינו מיותר
- תנאי הקדם הוא חלק בלתי נפרד של ממשק המודול ובפרט צריך להיות חלק מהתיעוד לכותבי צד הלקוח ('המשתמשים')
- על תנאי הקדם להיות בלתי תלוי במימוש



# מודולים סובלניים (tolerant)

- כאשר הספק בעצמו מתמודד עם מקרי הקצה, הוא מחליש את תנאי הקדם, אבל יוצר בלבול בתחומי האחריות וסיבוך בעיצוב המערכת
- מסננים (filters) חוצצים בין לקוחות לא זהירים ומודולים לא מוגנים
- בדוגמא הבאה STACK3 מגן על STACK2 מפני משתמשים לא זהירים
- השימוש בהאצלה (delegation) עשוי להיות מוחלף בירושה (inheritance)



# STACK3

```
template<class T>
class STACK3
{
public:
    /** Allocate stack for maximum of n elements,
     *  if n>=0. Otherwise set error to
     *  Negative_size. No precondition!
     */
    STACK3(int n):      representation(NULL),
                    error(NoError)
    {
        if(n>=0)
        {
            capacity = n;
            representation = new STACK2<T>(capacity);
        }
        else
            error = Negative_size;
    }
}
```



# STACK3

```
/**
 * @post: error_code_if_impossible:
 *         (n<0) == (error == Negative_size)
 * @post: no_error_if_possible:
 *         (n>=0) == (error == NoError)
 * @post: capacity_set_if_no_error:
 *         (error == NoError) => (capacity == n)
 * @post: allocated_if_no_error:
 *         (error == NoError) =>
 *         representation != NULL
 * @post: stack_empty_if_no_error:
 *         (error == NoError) => empty()
 */
STACK3(int n);
```



# STACK3

```
/** Top element if present, otherwise a new object with
 * the type's default value is allocated and returned, with
 * error set to Underflow. No precondition!
 * @post: error_code_if_impossible:
 *         ($prev(empty())) == (error == Underflow)
 * @post: no_error_if_possible:
 *         (!($prev(empty()))) == (error == NoError)
 */
T &top() const
{
    if (!empty())
    {
        assert(representation!=NULL);
        error = NoError;
        return representation->top();
    }
    else
        error=Underflow;
    return new T();
}
```





# STACK3

```
/** Is stack empty?  
 * @post: empty_definition: $result == (count() == 0)  
 */  
bool empty() const  
{  
    return (count() == 0) || representation->empty();  
}  
  
/** Is stack full?  
 * @post: full_definition: $result == (count == capacity)  
 */  
bool full() const  
{  
    return (count() == capacity) || representation->full();  
}
```



# STACK3

```
/** Add x on top if possible, otherwise set  
 * error to Overflow. No precondition!  
 */
```

```
void push(const T& x)  
{  
    if (full())  
        error = Overflow;  
    else  
    {  
        assert(representation != NULL);  
        representation->push(x);  
        error = NoError;  
    }  
}
```

```
}  
26
```



# STACK3

```
/** Add x on top if possible, otherwise set
 * @post: error_code_if_impossible:
 *     ($prev(full()) == (error == Overflow)
 * @post: no_error_if_possible:
 *     !($prev(full()) == (error == NoError)
 * @post: not_empty_if_no_error:
 *     (error==NoError) => !empty()
 * @post: added_to_top_if_no_error:
 *     (error==0) => top() == x
 * @post: one_more_item_if_no_error:
 *     (error==NoError) =>
 *         count() = $prev(count())+1
 */
void push(const T& x);
```



# STACK3

```
/** Remove top element if possible,  
 * otherwise set error to Underflow  
 * No precondition!  
 */  
void pop()  
{  
    if(empty())  
        error = Underflow;  
    else  
    {  
        asssert(representation != NULL);  
        representation->pop();  
        error = NoError;  
    }  
}
```



# STACK3

```
/**
 * @post: error_code_if_impossible:
 *       $prev(empty()) == (error == Underflow)
 * @post: no_error_if_possible:
 *       !($prev(empty())) == (error == NoError)
 * @post: not_full_if_no_error:
 *       (error==NoError) => !full()
 * @post: one_fewer_if_no_error:
 *       (error==NoError) => count() == $prev(count())-1
 */
void pop();

enum ErrorCode {NoError=0, Overflow, Underflow,
                Negative_size};

ErrorCode error;
```



# STACK3

```
private:
    int count() const
    {
        return representation->count;
    }

    /** unprotected stack used to hold elements
     */
    STACK2<T> *representation;

    /** The maximum number of stack elements
     */
    int capacity;
}
```



# שמורת המחלקה

# השמורה של המחלקה

## class invariant

- מאפיין של מחלקה
- ביטוי בולאני שערכו TRUE 'בכל רגע נתון'
- על האינוריאנטה להשתמר ע"י כל הפעולות המיוצאות של המחלקה
- דוגמא: השמורה של STACK2







# השמורה של STACK2

```
/**
 * @inv: count_non_negative: count >= 0
 * @inv: count_bounded: capacity >= count
 * @inv: consistent_with_array_size:
 *         capacity == representation.capacity()
 * @inv: empty_if_no_elements:
 *         empty() == (count == 0)
 * @inv: item_at_top:
 *         (count > 0) =>
 *         representation.at(count-1) == top()
 */
template<class T>
class STACK2 { ... }
```



# כלל השמורה



NO CAMPING

ביטוי I הוא שמורת מחלקה נכונה עבור מחלקה C אם מתקיימים שני התנאים הבאים:

1. עבור כל הפעלה של בנאי של C עם ארגומנטים המקיימים את התנאי המקדים של פונקציית הבנאי, במצב שבו לכל השדות יש את ערכי ברירת המחדל שלהם, מתקבל בסופו של הבנאי מצב המקיים את I

2. עבור כל הפעלה של מתודה מיוצאת של המחלקה עם ארגומנטים המקיימים את התנאי המקדים שלה במצב המקיים הן את I והן את תנאי הקדם של המתודה, מתקבל מצב המקיים את I



# כלל השמורה



- לכל מחלקה יש בנאי, אם לא הוגדר כזה נקרא בנאי ברירת המחדל (הבנאי הריק חסר הארגומנטים)
- מצב של מחלקה מוגדר להיות אוסף ערכי כל השדות של המחלקה
- תנאי קדם עשוי להתייחס הן למצב של המחלקה והן לערכי הארגומנטים
- תנאי בתר עשוי להתייחס למצב המחלקה בסוף פעולת הפונקציה, למצב לפני פעולת הפונקציה (תוך שימוש ב `$prev`) וכן לערך המוחזר של הפונקציה (אם קיים, בעזרת `$result`)
- השמורה יכולה להתייחס רק למצב של המחלקה



# כלל השמורה



- אף על פי שתנאי הקדם, הבתר והשמורה מכילים התייחסות לפונקציות, זו בעצם התייחסות עקיפה לשדות
- ניתן להתייחס אל השמורה כתנאי קדם ותנאי בתר של כל אחת מהמתודות המיוצאות של המחלקה וכן כתנאי בתר של הבנאי
- השמורה מייצגת אחריות הן של הספק בלבד (כמו במקרה של תנאי בתר)



# נכונות של מחלקה

○ מחלקה C היא נכונה (לפי החוזה שלה) אם מתקיים:

● בעבור כל רוטינה מיוצאת R, ואוסף ארגומנטים חוקיים  $x_R$  מתקיים:

$$\{INV \ \& \ pre_R(x_R)\} B_R \{INV \ \& \ post_R\}$$

# נכונות של מחלקה

- בעבור כל בנאי  $P$ , ואוסף ארגומנטים חוקיים  $x_P$  מתקיים:

$$\{Defaults_C \& pre_P(x_P)\} B_P \{INV \& post_P\}$$

כאשר:

- $Defaults_C$  הם ערכי ברירת המחדל של שדות  $C$
- $INV$  היא השמורה של  $C$



# תפקיד הבנאי



- לאחר שנקבעה השמורה של המחלקה, תפקידו של הבנאי להביא את המחלקה למצב שהיא מקיימת את השמורה שלה
- לדוגמא: השמורה והבנאי של המחלקה  $shifted\_vector<T>$  - מחלקה המספקת מערך בעל תחום אינדקסים  $min$  עד  $max$  (אשר מקרה פרטי שלו זה 0 עד  $capacity-1$ )



# *shifted\_vector*<T>

```
template <class T>
class shifted_vector
{
public:
    /** Allocate array with bounds
     * from `minindex' to `maxindex`
     * (empty if `minindex' > `maxindex')
     * @inv: consistent_count:
     *       count == upper - lower + 1
     * @inv: non_negative_count: count >= 0
     */
    shifted_vector(int minindex, int maxindex)
    {...}
}
```





# *shifted\_vector*<T>

```
/**
 * @pre: meaningful_bounds: maxindex>=minindex-1
 * @post: exact_bounds_if_non_empty:
 *   (maxindex>=minindex) =>
 *   ((lower==minindex) && (upper==maxindex))
 * @post: conventions_if_empty:
 *   (maxindex<minindex) =>
 *   ((lower==1) && (upper==0))
 */
shifted_vector(int minindex, int maxindex)
{...}
```



# *shifted\_vector*<T>

```
/** Entry at index `i`, if in interval
 * @pre: index_not_too_small: lower<=i
 * @pre: index_not_too_large: i<=upper
 */
T &operator[](int i) {...}
```



## *shifted\_vector*<T>

```
/** Assign v to the entry at index `i`,  
 *   if in interval  
 * @pre: index_not_too_small: lower<=i  
 * @pre: index_not_too_large: i<=upper  
 * @post: element_replaced: item(i) == v  
 */
```

```
void put(T& v, int i) {...}
```

```
private:
```

```
/** Minimum, maximum index; array size */
```

```
int lower, upper, count;
```

```
...
```

```
};
```

# השמורה וטיפוס הנתונים המופשט

○ ראינו כי הגדרת טיפוס נתונים מופשט כוללת 4 חלקים:



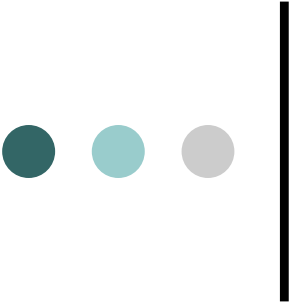
- שם הטיפוס והפרמטרים הכלליים
- חתימת הפונקציות
- תנאים מקדימים לפונקציות
- אקסיומות

○ נרצה לממש מחלקות לפי 4 החלקים המרכיבים את הטיפוס המופשט שאותו הן ממשות



# סיווג הפעולות על ADT תזכורת

- כאשר  $f$  היא פונקציה של טיפוס מופשט  $T$  מהצורה  
 $f : A \times B \dots \rightarrow x$ , אזי  $f$  היא:
  - **שאילתה** (query, accessor) אם  $T$  מופיע רק בצד שמאל. ואז  $f$  תהיה שדה של המחלקה או מתודה שלה
  - **פקודה** (transformer, command) אם  $f$  מופיעה משני צידי החץ. ואז  $f$  תהיה מתודת `void` (פרוצדורה) המשנה את המצב של המחלקה
  - **בנאי** (constructor, creator) אם  $T$  מופיעה רק בצד ימין של החץ. ואז  $f$  תהיה בנאי של המחלקה



# תנאים מקדימים ואקסיומות

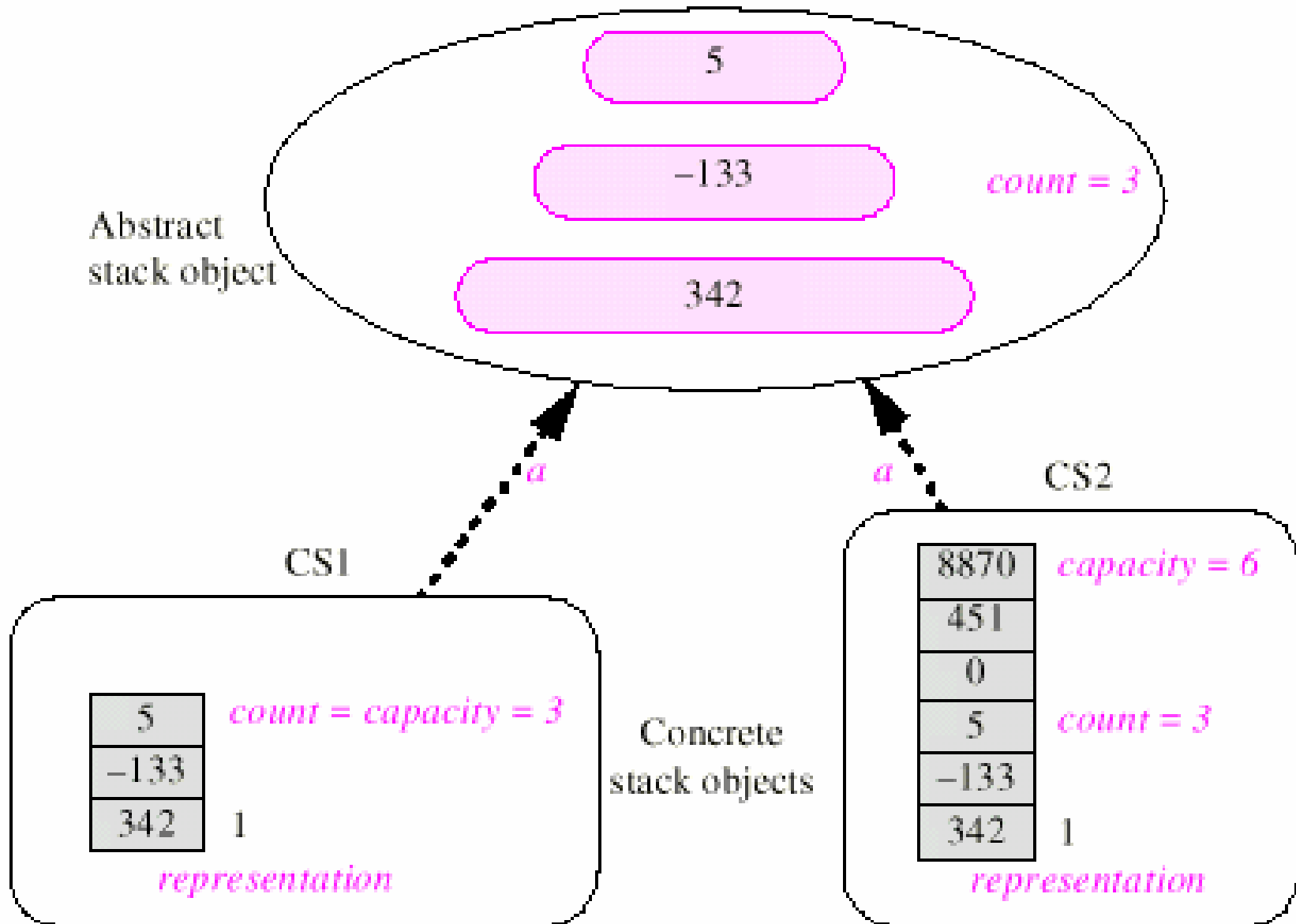
- תנאי מקדים בטיפוס המופשט **יופיע** כתנאי מקדים של אחת המתודות של המחלקה
- אקסיומות המכילות פקודות (אולי בשילוב שאילתות) **יופיעו** כתנאי הבתר של הפרוצדורות המתאימות
- אקסיומות המכילות רק שאילתות **יופיעו** כתנאי בתר של המתודות המתאימות או (כאשר הן מכילות יותר משאילתה אחת ולפחות אחת מהן מומשה כשדה) כחלק מהשמורה
- אקסיומות המכילות בנאים **יופיעו** בתנאי הבתר של אותם בנאים

# שמורות וטיפוסים מופשטים



- נשווה בין האקסיומות של STACK והמחלקה STACK2
- לא כל אקסיומה ניתן לבטא ע"י שפת הטענות הבולאניות (assertions). ניתן להשתמש בהערות מילוליות
- כמה מהטענות של החוזה אינן לקוחות מהאקסיומות כגון:  $0 \leq \text{count}$  או  $\text{count} \leq \text{capacity}$
- שמורות מימוש מבטאות עקביות בין המימוש ובין הטיפוס המופשט

# פונקציית ההפשטה





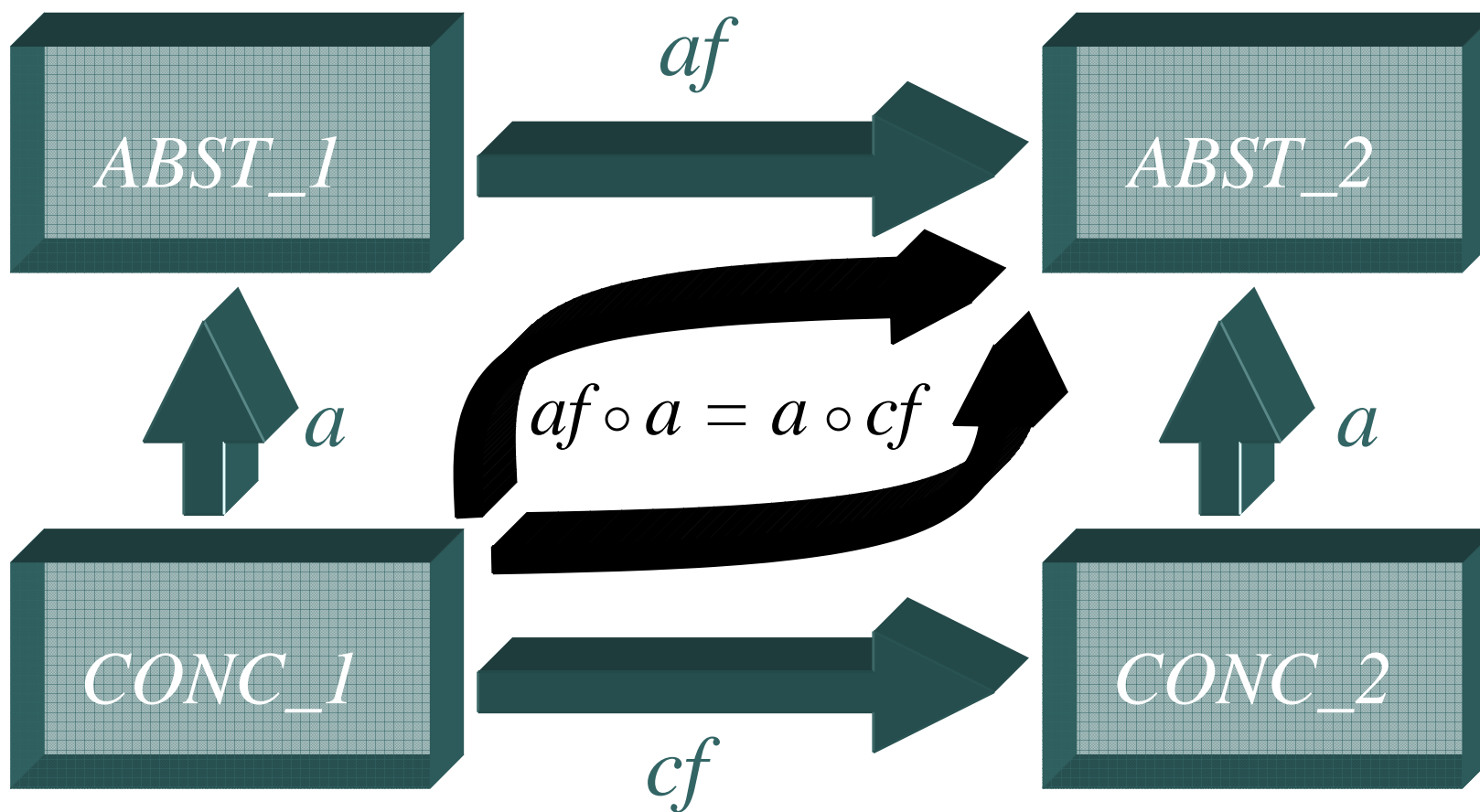


# פונקציית ההפשטה

- פונקציית ההפשטה אינה חייבת להיות חד-חד ערכית. בדוגמא (STACK2) רואים שני עצמים במצב גשמי (concrete state) שונה הממופים לאותו מצב מופשט
- פונקציית ההפשטה אינה חייבת להיות מלאה. לא כל מצב גשמי של הייצוג מתאים למצב מופשט. לדוגמא: רק עצמים המקיימים את השמורה  $0 \leq count; count \leq capacity$  ממופים למצב מופשט

# נכונות המימוש

Abstract objects (instances of the ADT  $A$ )



Concrete objects (instances of the class  $C$ )

# שימוש ב assert

```
#include<cassert>
void assert(boolean_expression);
```

- משמשת לציין במפורש (שלא ע"י הערה) כי תנאי מסוים חייב להתקיים אחרת זוהי טעות מימוש לוגית
- לדוגמא בפונקציית push של STACK3:

```
assert(representation != NULL);
representation->push(x);
error = NoError;
```

- השימוש ב assert מקובל אצל הלקוח שרוצה להצדיק את הלגיטימיות של הקריאה שהוא עומד לבצע

# שמורת הלולאה

```
/**
```

```
* @loop_variant: loop_variant
```

```
* @loop_invariant: loop_invariant
```

```
*/
```

```
initialization_instructions
```

```
while (as_long_as)
```

```
loop_instructions
```

מספר שלם אי שלילי  
שערכו יורד בכל מחזור  
של הלולאה

אולי ריק

ביטוי לוגי  
(assertion) שערכו  
תמיד נכון



# המחלק המשותף הגדול ביותר

```
/** Greatest common divisor of a and b
 * @pre: a>0
 * @pre: b>0
 * @post: returns the gcd of a and b
 */
int gcd(int a,b)
{
    int x, y;
    ... see next slide
}
```



# המחלק המשותף הגדול ביותר

```
/**
 * @loop_variant: max(x,y)
 * @loop_invariant: x>0 && y>0
 * @loop_invariant: (x,y) have same gcd as (a,b)
 */
x=a; y=b;
while(x!=y)
    if (x>y)
        x=x-y;
    else
        y=y-x;
```



# הוספת טענות לקוד

- **עוזרת לכתוב קוד נכון:** תכנות ע"פ חוזה
- **עוזרת לתיעוד:** עדיפה על הערות מילוליות. שימוש בכלי עזר (כגון: javadoc ל java , short ל- Eiffel , ורבים אחרים ל C++) כדי להוציא מהמחלקה את המידע הרלוונטי ללקוח (רק נתונים מיוצאים, רק ממשק ללא גופי פונקציות)
- **עוזרת לבדיקת הקוד,** ניפוי שגיאות ובקרת איכות
- **מאפשרת לתוכנה סובלנות משתנה**

# doc++ 1 STACK2



**Public Fields**

- const int **capacity**  
*The maximum number of stack elements*

**Public Methods**

- T **top()**  
*Top element @pre: not\_empty: !empty()*
- bool **empty()**  
*Is stack empty?*
- bool **full()**  
*Is stack full?*
- void **push**(T x)  
*Add x on top @pre: not\_full: !full() @post: not\_empty: !empty() @post: added\_to\_top: top() = (count)+1 @post: in\_top\_array\_entry: representation[count] == x*
- void **pop()**  
*@pre: not\_empty: !empty() @post: not\_full: !full() @post: one\_fewer: count == \$prev(count)*





# ניטור בזמן ריצה

- מכיוון של C++ אין מנגנון מובנה לאכיפה של טענות (פרט ל assert של שפת C) אכיפת החוזה ושאר הטענות תלויה בכלים חיצוניים
- אחת הגישות היא להשתמש במחולל קוד, אשר קורא את כל הטענות המופיעות בקוד המקור (חוזים, assert, שמורות לולאה – גם אם הן מופיעות כהערות) ומייצר עבורן קוד שיאכוף אותן בזמן ריצה
- כלי כזה עשוי להגדיר רמות אכיפה נדרשות (ב Eiffel יש 6 רמות) לכל מחלקה או מודול, ורק טענות המתאימות לרמת האכיפה יאכפו בזמן ריצה
- ב C++ הוספת השורה `#define NDEBUG` לפני שורת `#include<cassert>` תבטל את בדיקת ה `assert` בזמן ריצה



# ניטור בזמן ריצה דיון

- בטיחות מול יעילות
- שלב הפיתוח מול שלב היצור ( debug vs. release )
- חריגה מגבולות מערך
- בשפת Eiffel בדיקת תנאי הקדם בלבד גורמת לתקורה של 50% בביצועים



# שילוב טענות בקוד דיון

- ניטור לעומת אימות
- כח ביטוי – לא כל טענה לוגית ניתן לבטא בשפת התכנות, לדוגמא:  $pop(push(x,s))=s$
- פונקציות המופיעות בתוך טענות:
  - האם הפונקציות אינן משנות את המצב של התוכנית (no side effect)
  - האם הפונקציות עצמן נכונות
  - מי מספר את הספר?
- שמורות ומצביעים – מתי נבדוק שינוי בעצם מוצבע?

# שמורות ומצביעים

```
/** Indirect invariant effect
 * @inv: forward==NULL ||
 *      (forward->backward == this)
 */
class A {
    B *forward;

public:
    A() : forward(NULL) {}

    /** Chain b1 to current instance */
    void attach(B *b1)
    {
        forward = b1; // Update b1's backward
                     // reference for consistency
        if (b1 != NULL)
            b1->attach(this);
    }
};
```



# שמורות ומצביעים

```
class B
{
    A *backward;

public:
    B() : backward(NULL) {}

    /** Chain a1 to current instance
     */
    void attach(A *a1)
    {
        backward = a1;
    }
};
```



# שמורות ומצביעים

```
int main()  
{  
    A a1, a2;  
    B b1;  
  
    a1.attach(&b1)  
    b1.attach(NULL) // A's invariant  
                    // violation  
    a2.attach(b1) // A's invariant  
                  // violation  
}
```