

# תכנות מונחה עצמים בשפת C++

אוהד ברזילי  
אוניברסיטת תל אביב

# תבניות תיכון - המשך (Design Patterns)

המצגת מבוססת על הספר:

Design Patterns: Elements of Reusable Object-Oriented Software  
By Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

כל הזכויות שמורות למחברים

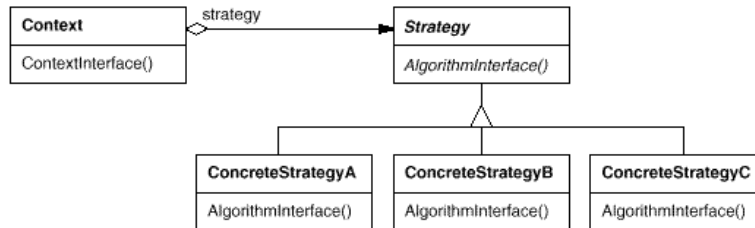
## סיווג תבניות

Scope	Class	Purpose		
		Creational	Structural	Behavioral
	Class	<a href="#">Factory Method (107)</a>	<a href="#">Adapter (139)</a>	<a href="#">Interpreter (243)</a> <a href="#">Template Method (325)</a>
	Object	<a href="#">Abstract Factory (87)</a> <a href="#">Builder (97)</a> <a href="#">Prototype (117)</a> <a href="#">Singleton (127)</a>	<a href="#">Adapter (139)</a> <a href="#">Bridge (151)</a> <a href="#">Composite (163)</a> <a href="#">Decorator (175)</a> <a href="#">Facade (185)</a> <a href="#">Proxy (207)</a>	<a href="#">Chain of Responsibility (223)</a> <a href="#">Command (233)</a> <a href="#">Iterator (257)</a> <a href="#">Mediator (273)</a> <a href="#">Memento (283)</a> <a href="#">Flyweight (195)</a> <a href="#">Observer (293)</a> <a href="#">State (305)</a> <a href="#">Strategy (315)</a> <a href="#">Visitor (331)</a>

## תבנית התיכון Strategy

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

## תבנית התיכון Strategy - מבנה



תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

5

## תבנית התיכון Strategy השפעה על קוד התוכנית

■ החלפה 'נאיבית' של אלגוריתמים:

```
void Composition::Repair () {
    switch (_breakingStrategy) {
    case SimpleStrategy:
        ComposeWithSimpleCompositor();
        break;
    case TeXStrategy:
        ComposeWithTeXCompositor();
        break;
    // ...
    }
    // merge results with existing composition, if
    // necessary
}
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

6

## תבנית התיכון Strategy השפעה על קוד התוכנית

- התבנית strategy מייתרת את ה  
:switch

```
void Composition::Repair ()  
{  
    _compositor->Compose();  
    // merge results with existing  
    // composition, if necessary  
}
```

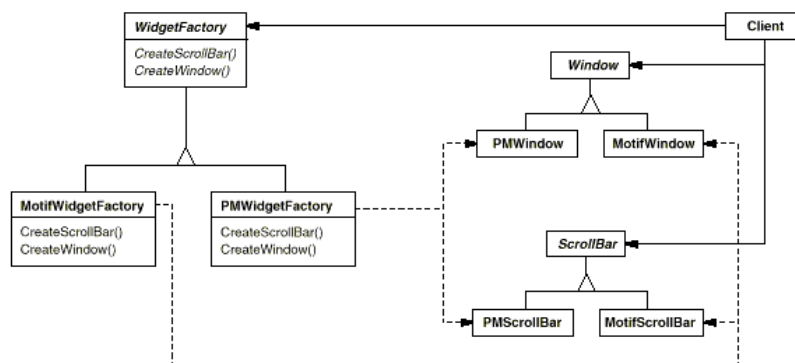
## תבנית התיכון Strategy – משתתפים

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

## תבנית התיכון Abstract Factory

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

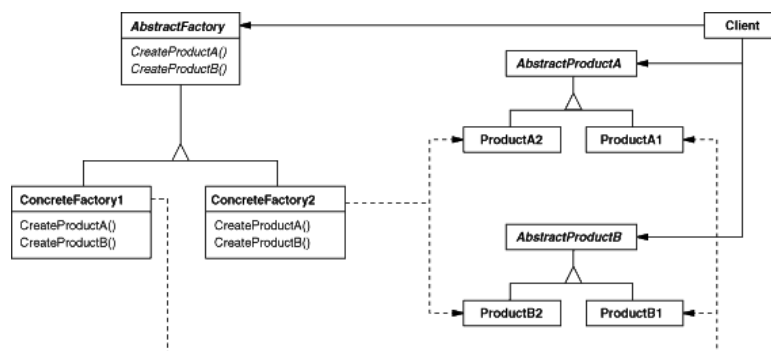
## תבנית תיכון Abstract Factory - דוגמא



## תבנית תיכון Abstract Factory - ישימות

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

## תבנית תיכון Abstract Factory - מבנה



## תבנית תיכון Abstract Factory משתתפים

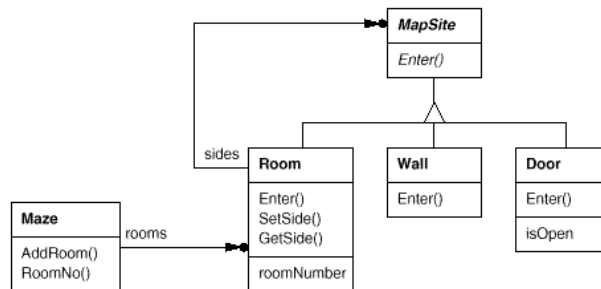
- AbstractFactory
  - מגדיר שרותים מופשטים ליצירת כל המוצרים המופשטים.
- ConcreteFactory (N גרסאות)
  - יורש מ AbstractFactory ומיישם את השרותים ליצירת מוצרים מוחשיים.
- AbstractProduct (M גרסאות)
  - מגדיר מנשק לטיפוס של מוצר.
- ConcreteProduct (N\*M גרסאות)
  - יורש מ AbstractProduct ומיישם את הממשק.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

## תבנית תיכון Abstract Factory משתתפים (המשך)

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

## דוגמא

■ נניח שאנו מתכננים משחק מבוך:



```
enum Direction {North, South, East, West};
```

## דוגמא - מבוך

```
class Room : public MapSite
{
public:
    Room(int roomNo);
    MapSite* GetSide(Direction) const;
    void SetSide(Direction, MapSite*);
    virtual void Enter();

private:
    MapSite* _sides[4];
    int _roomNumber;
};
```



## דוגמא - מבוך

```
class Wall : public MapSite
{
public:
    Wall();

    virtual void Enter();
};
```

```
class Door : public MapSite
{
public:
    Door(Room* = 0, Room* = 0);
    virtual void Enter();
    Room* OtherSideFrom(Room*);

private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

## דוגמא - מבוך

- המחלקה Maze משמשת מיכל ל MapSites
- כדי לתאר משחק מסוים נגדיר את מהחלקה MazeGame שתתחל Maze בהרכב מסוים של חדרים ודלתות

## דוגמא - מבוך

```
Maze* MazeGame::CreateMaze ()
{
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door* theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);

    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

## דוגמא - מבוך

- המתודה CreateMaze לא מתאימה להתפתחות עתידית של משחק המבוך
- אם נרצה בעתיד להוסיף אולי EnchantedMaze שיתמוך בדלתות מכושפות או BombedMaze שיתמוך בחדרים ממולכדים נצטרך להחליף את כל הפונקציה מכיוון שהיא יוצרת מפורשות חדרים ודלתות פשוטים
- הוספת AbstractMazeFactory מכילה את תהליך יצירת המבוך לתמיכה במשפחות מוצרים עתידיות

## מפעל למבוכים

```
class MazeFactory {
public:
    MazeFactory();

    virtual Maze* MakeMaze() const
    { return new Maze; }

    virtual Wall* MakeWall() const
    { return new Wall; }

    virtual Room* MakeRoom(int n) const
    { return new Room(n); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new Door(r1, r2); }
};
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

21

## מבוך עם מפעל

```
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
    Maze* aMaze = factory.MakeMaze();
    Room* r1 = factory.MakeRoom(1);
    Room* r2 = factory.MakeRoom(2);
    Door* aDoor = factory.MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, factory.MakeWall());
    r1->SetSide(East, aDoor);
    r1->SetSide(South, factory.MakeWall());
    r1->SetSide(West, factory.MakeWall());
}
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

22

## מבוך עם מפעל

```
r2->SetSide(North, factory.MakeWall());
r2->SetSide(East, factory.MakeWall());
r2->SetSide(South, factory.MakeWall());
r2->SetSide(West, aDoor);

return aMaze;
}
```

- נשווה את יצירת המבוך עם המבוך המקורי
- פתרון זה הוא כללי יותר מכיוון שבפועל ניתן להעביר לפונקציה מפעל מכושף במקום המפעל המופשט -

## מפעל מכושף

```
class EnchantedMazeFactory : public MazeFactory {
public:
    EnchantedMazeFactory();

    virtual Room* MakeRoom(int n) const
    { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
    { return new DoorNeedingSpell(r1, r2); }

protected:
    Spell* CastSpell() const;
};
```

## מפעל ממולכד

- או מפעל למבוך ממולכד:

```
Wall* BombedMazeFactory::MakeWall () const  
{ return new BombedWall; }
```

```
Room* BombedMazeFactory::MakeRoom(int n) const  
{ return new RoomWithABomb(n); }
```

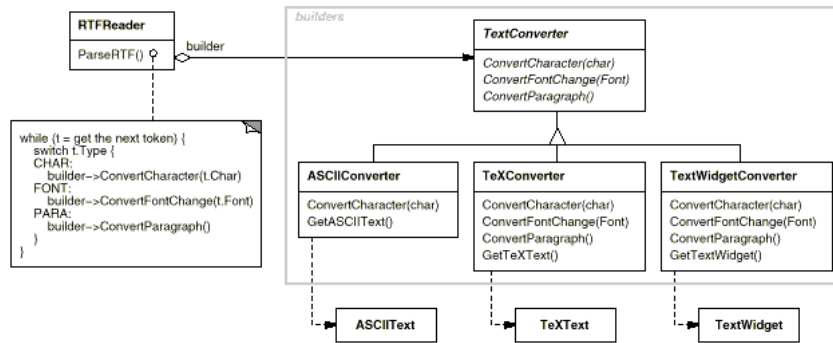
- אתחול משחק יראה כך:

```
MazeGame game;  
BombedMazeFactory factory;  
  
game.CreateMaze(factory);
```

## Builder

- הפרדת תהליך יצירה של עצם מורכב מהייצוג שלו, כך שניתן לחזור על אותו תהליך יצירה עבור עצמים ממחלקות שונות
- לדוגמא: המרת מסמך מפורמט אחד למשנהו

# המרת פורמט בעזרת Builder

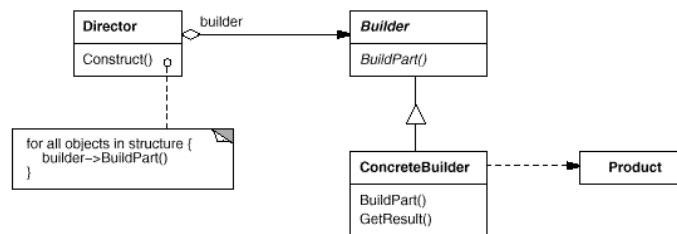


```

while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(L.Char)
    FONT:
        builder->ConvertFontChange(L.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
    
```

# Builder

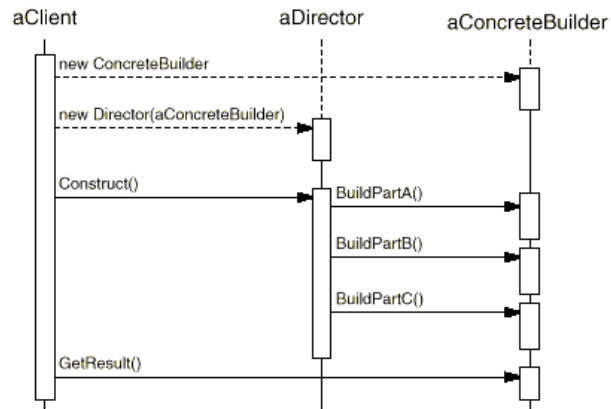
■ או בהכללה:



```

for all objects in structure {
    builder->BuildPart()
}
    
```

# Builder



תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

29

## דוגמא – בונה מבוכים

```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }
    virtual Maze* GetMaze()
    { return 0; }

protected:
    MazeBuilder();
};
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

30

## דוגמא – בונה מבוכים

```
Maze* MazeGame::CreateMaze (MazeBuilder& builder)
{
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```

## ערך מוסף

- בעזרת בונה המבוכים הצלחנו להפריד את תהליך בניית המבוך ממבנה המבוך בפונקציה CreateMaze
- את התהליך נגדיר ואולי אף נשכלל בעתיד על ידי הבונה ומזרותיו כגון: StandardMazeBuilder



## דוגמא – בונה מבוכים

```
class StandardMazeBuilder : public MazeBuilder {
public:
    StandardMazeBuilder();
    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual Maze* GetMaze();

private:
    Direction CommonWall(Room*, Room*);
    Maze* _currentMaze;
};
```

## דוגמא – בונה מבוכים

```
StandardMazeBuilder::StandardMazeBuilder ()
{ _currentMaze = 0; }

void StandardMazeBuilder::BuildMaze ()
{ _currentMaze = new Maze; }

Maze* StandardMazeBuilder::GetMaze ()
{ return _currentMaze; }
```

## דוגמא – בונה מבוכים

```
void StandardMazeBuilder::BuildRoom (int n)
{
    if (!_currentMaze->RoomNo(n))
    {
        Room* room = new Room(n);
        _currentMaze->AddRoom(room);
        room->SetSide(North, new Wall);
        room->SetSide(South, new Wall);
        room->SetSide(East, new Wall);
        room->SetSide(West, new Wall);
    }
}
```

## דוגמא – בונה מבוכים

```
void StandardMazeBuilder::BuildDoor (int n1, int n2)
{
    Room* r1 = _currentMaze->RoomNo(n1);
    Room* r2 = _currentMaze->RoomNo(n2);

    Door* d = new Door(r1, r2);
    r1->SetSide(CommonWall(r1,r2), d);
    r2->SetSide(CommonWall(r2,r1), d);
}
```

■ וכדי ליצור משחק חדש:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;

game.CreateMaze(builder);
maze = builder.GetMaze();
```

## בונה-סופר

- הפרדת המבנה מהתהליך מאפשרת הגדרה של בונה-סופר המדפיס את מספר רכיבי המבוך:

```
class CountingMazeBuilder : public MazeBuilder {
public:
    CountingMazeBuilder();
    virtual void BuildMaze();
    virtual void BuildRoom(int);
    virtual void BuildDoor(int, int);
    virtual void AddWall(int, Direction);
    void GetCounts(int&, int&) const;

private:
    int _doors;
    int _rooms;
};
```

## בונה-סופר

```
CountingMazeBuilder::CountingMazeBuilder ()
{ _rooms = _doors = 0; }

void CountingMazeBuilder::BuildRoom (int)
{ _rooms++; }

void CountingMazeBuilder::BuildDoor (int, int)
{ _doors++; }

void CountingMazeBuilder::GetCounts(int& rooms,
int& doors ) const
{ rooms = _rooms; doors = _doors; }
```

## בונה-סופר

■ השימוש בבונה-סופר זהה לשימוש בכל בונה אחר:

```
int rooms, doors;
MazeGame game;
CountingMazeBuilder builder;

game.CreateMaze(builder);
builder.GetCounts(rooms, doors);
cout << "The maze has " << rooms
      << " rooms and " << doors << " doors" << endl;
```

## Singleton

■ יצירת מחלקה עם מופע אחד בלבד:

```
class Singleton {
public:
    static Singleton* Instance();

protected:
    Singleton();

private:
    static Singleton* _instance;
};
```

# Singleton

■ ובקובץ המימוש:

```
Singleton* Singleton::_instance = 0;
```

```
Singleton* Singleton::Instance ()  
{  
    if (_instance == 0) {  
        _instance = new Singleton;  
    }  
    return _instance;  
}
```

■ מה קורה אם יש כמה טיפוסים אפשריים ל Singleton איך נדע איזה טיפוס ליצור?

# Singleton

■ נתחזק מיפוי של הטיפוסים האפשריים:

```
class Singleton {  
public:  
    static void Register(const char* name, Singleton*);  
    static Singleton* Instance();  
  
protected:  
    static Singleton* Lookup(const char* name);  
  
private:  
    static Singleton* _instance;  
    static List<NameSingletonPair>* _registry;  
};
```

# Singleton

- בעת היצירה \ גישה נבדוק מול המיפוי:

```
Singleton* Singleton::Instance ()
{
    if (_instance == 0)
    {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

# Singleton פולימורפי

- והיכן ה Singleton רושם את עצמו? אפשרות אחת היא בבנאי:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

- כמובן, שהדבר סותר את רעיון ה Singleton
- יש צורך לאתחל מפורשות מופע static אחד של כל טיפוס ובכך להשאיר ל Singleton רק את ניהול הגישה אליו ולא את היצירה שלו

## Singleton - MazeFactory

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here

protected:
    MazeFactory();

private:
    static MazeFactory* _instance;
};

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

## Singleton - MazeFactory עם יורשים

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;
        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            instance = new EnchantedMazeFactory;
        }

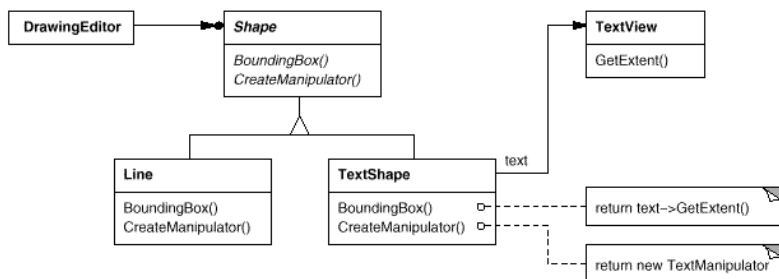
        // ... other possible subclasses

    } else { // default
        _instance = new MazeFactory;
    }
    return _instance;
}
```

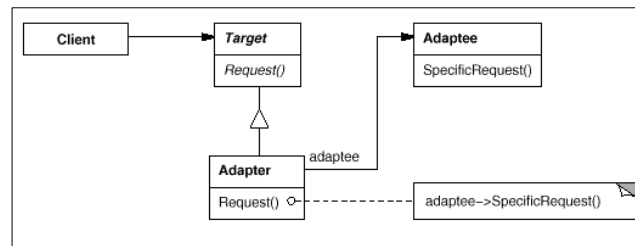
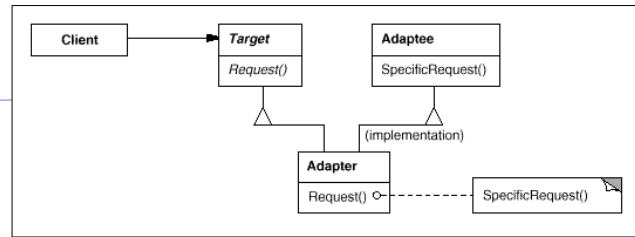
## Adapter (תבניות מבנה)

- התאמת ממשק של מחלקה לממשק שלו מצפים הלקוחות שלה
- לדוגמא: מחלקה להצגת טקסט המעוניינת להיות גם תת מחלקה של Shape כדי להיות מוצגת על המסך וגם להשתמש בלוגיקה המוגדרת ב TextView שתי גרסאות:
- Class Adapter – ירושה מרובה (בדר"כ אחת private)
- Object Adapter – הפנייה לעצם המספק את השרות

## Object Adapter







## דוגמת קוד

```

class Shape {
public:
    Shape();
    virtual void BoundingBox( Point& bottomLeft,
                             Point& topRight ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};
  
```

## דוגמת קוד

```
class TextShape : public Shape, private TextView
{
public:
    TextShape();
    virtual void BoundingBox( Point& bottomLeft,
        Point& topRight ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};
```

## דוגמת קוד

```
void TextShape::BoundingBox ( Point& bottomLeft,
    Point& topRight ) const
{
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}
```

## דוגמת קוד

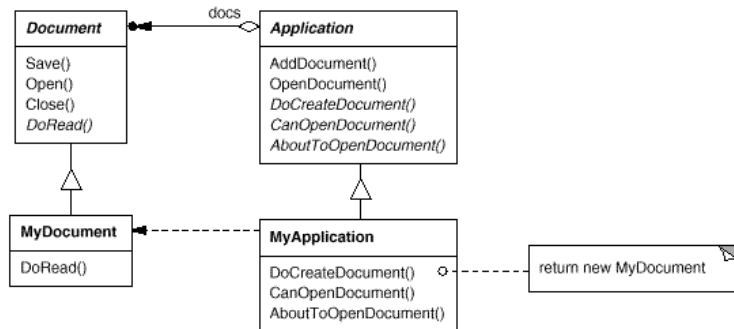
```
bool TextShape::IsEmpty () const
{
    return TextView::IsEmpty();
}

// Factory Method
Manipulator* TextShape::CreateManipulator () const
{
    return new TextManipulator(this);
}
```

## Template Method (תבניות התנהגות)

- הגדרה שלד אלגוריתם תוך דחיית כמה צעדים למחלקות נגזרת. מחלקות נגזרת יכולות להגדיר מחדש מקטעים ללא צורך להגדיר מחדש את כל האלגוריתם
- לדוגמא: פתיחת מסמך ביישום ללא תלות בסוג המסמך או היישום בפועל

## Template Method



תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

55

## פתיחת מסמך כאלגוריתם כללי

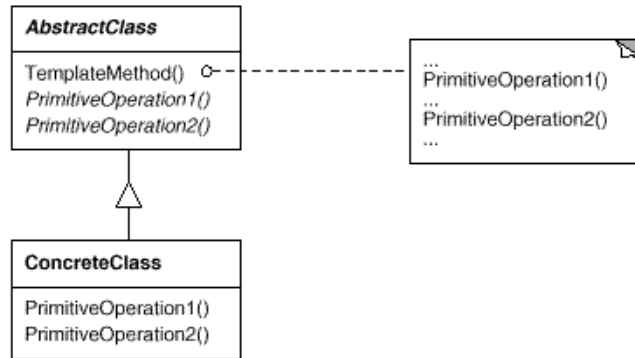
```
void Application::OpenDocument (const char* name)
{
    if (!CanOpenDocument(name)) {
        return; // cannot handle this document
    }

    Document* doc = DoCreateDocument();
    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

56

# Template Method



# Template Method

- סוגי הפעולות הנקראות ממתודת תבנית:
  - concrete operations – מתודות של מחלקות אחרות (לקוחות, ספקים)
  - concrete AbstractClass operations – מתודות שיהיו שימושיות גם עבור היורשים
  - primitive operations – פעולות בסיסיות, אבני בניין לשימוש מתודות כלליות
  - factory methods – יצירת עצמים ללא ציון טיפוס המפורש
  - **hook operations** – התנהגות ברירת מחדל אשר מחלקות נגזרות מוזמנות להרחיב

## Hooks vs. Template Methods

■ הרחבת hook:

```
void DerivedClass::Operation () {  
    // DerivedClass extended behavior  
    ParentClass::Operation();  
}
```

■ נהפוך את Operation ל- Template Method:

```
void ParentClass::Operation () {  
    // ParentClass behavior  
    HookOperation();  
}  
  
void ParentClass::HookOperation () { }
```

## Hooks vs. Template Methods

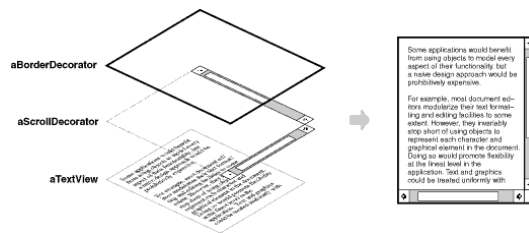
■ כעת יורש פוטנציאלי לא ישכח לקרוא למתודה המקורית, כי עליו לממש רק את ה-hook:

```
void DerivedClass::HookOperation () {  
    // derived class extension  
}
```

■ מחלקה צריכה להבהיר אילו מהמתודות שלה מופשטות (מחייבות הגדרה) ואילו הן hooks (מאפשרות הרחבה)

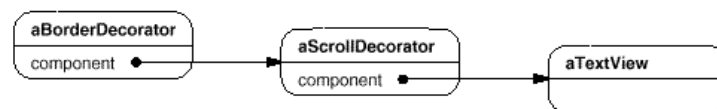
# Decorator (תבניות מבנה)

- הוספה דינאמית של תכונות שלא על ידי ירושה
- לדוגמא: מסמך עם מסגרת ופס גלילה

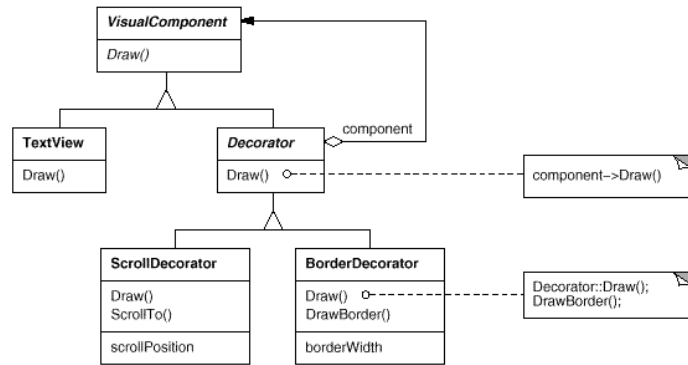


# Decorator

- הרעיון פשוט: כדי להוסיף תכונה לעצם ממחלקה קיימת נצביע עליו מעצם של מחלקה המממשת את התכונה הדרושה



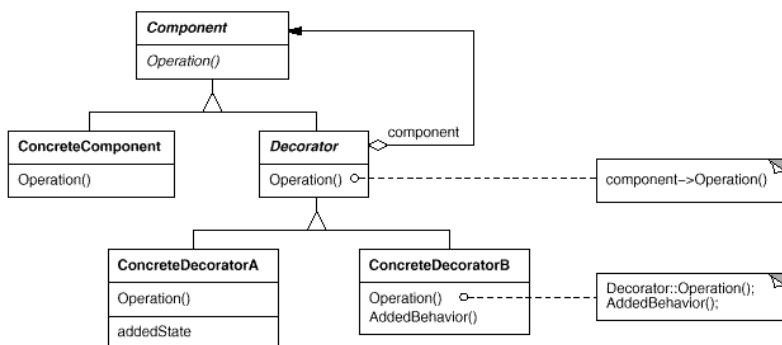
# Decorator



תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

63

# Decorator – המקרה הכללי



תכנות מונחה עצמים בשפת C++  
אוניברסיטת תל אביב

64



## דוגמת קוד

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

## דוגמת קוד

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw()
    { _component->Draw(); }

    virtual void Resize();
    { _component->Resize(); }

    // ...

private:
    VisualComponent* _component;
};
```

## דוגמת קוד

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();

private:
    void DrawBorder(int);

private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

## הרכבה

■ כעת קל להרכיב עצמים בעלי תכונות משתנות:

```
void Window::SetContents (VisualComponent* contents);

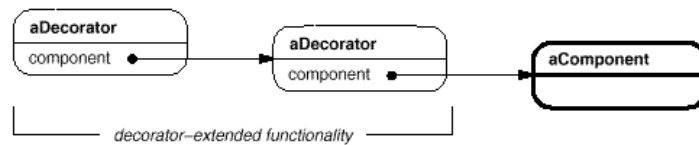
Window* window = new Window;
TextView* textView = new TextView;

window->SetContents(textView);           // Adds undecorated text

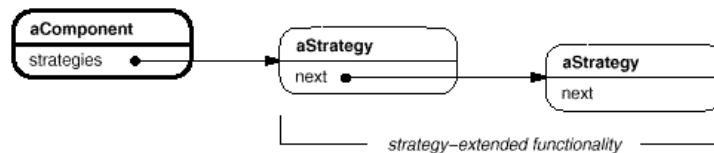
window->SetContents(                     // Adds decorated text
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

## Strategy vs. Decorator

### Decorator - החלפת המעטפת



### Strategy - הרחבת הליבה



תמונת מונחי עצמים בשפת C++  
אוניברסיטת תל אביב

69

## Strategy vs. Decorator

- נשים לב ש Strategy עשוי לגרום לשינוי ברכיב עצמו בעוד שב Decorator הרכיב נשאר אדיש
- מאידך ל Strategy יש ממשק משלו בעוד שה Decorator צריך לרשת מההורה של הרכיב
- לדוגמא: Strategy לצורך מסגרת צריך להגדיר ממשק לכך: `DrawBorder`, `GetWidth...` וכך גם אם הרכיב עצמו מסורבל – Strategy נשאר קליל

תכנות מונחי עצמים בשפת C++  
אוניברסיטת תל אביב

70

## סיכום תבניות תיכון

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