6. Шаблон Композиция (Composite)

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

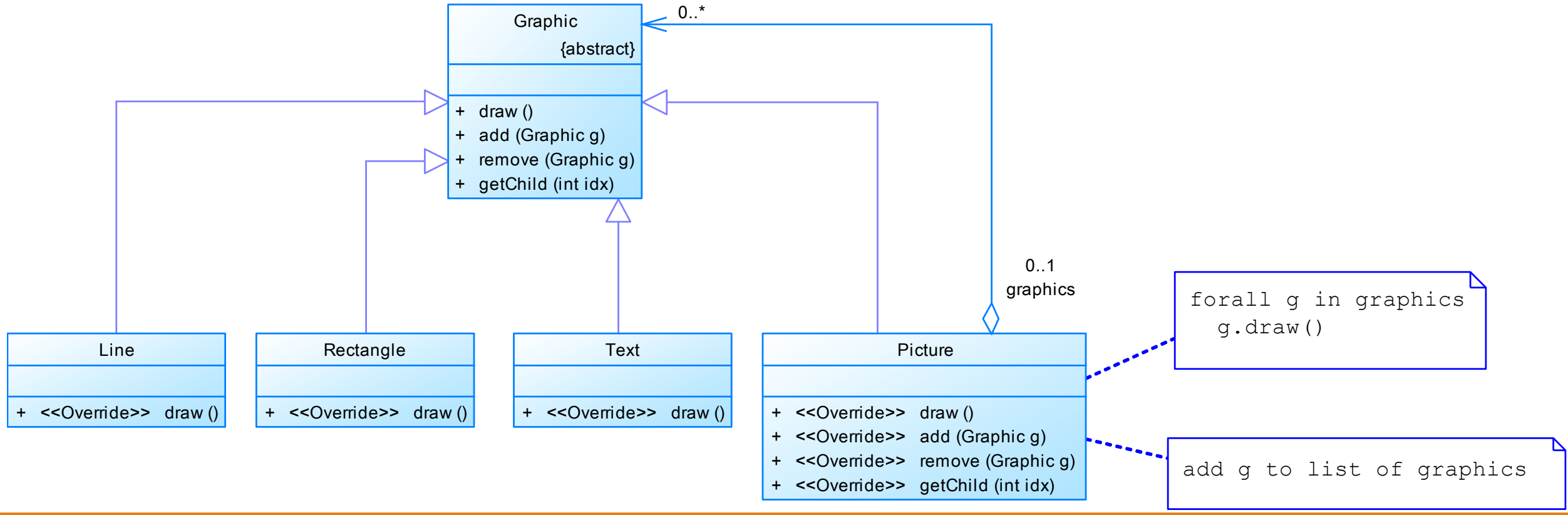
ДОЦ. Д-Р ЕМИЛ ДОЙЧЕВ

Общи сведения

- ✓ **Вид:** Структурен за обекти
- ✓ **Цел:** Композиране на обекти в дървовидни структури за представяне на йерархии от елементи. Шаблонът *Композиция* дава възможност на клиентите да третират еднакво отделни обекти и композиции от обекти.
- ✓ **Известен и като:** *няма алтернативни имена*

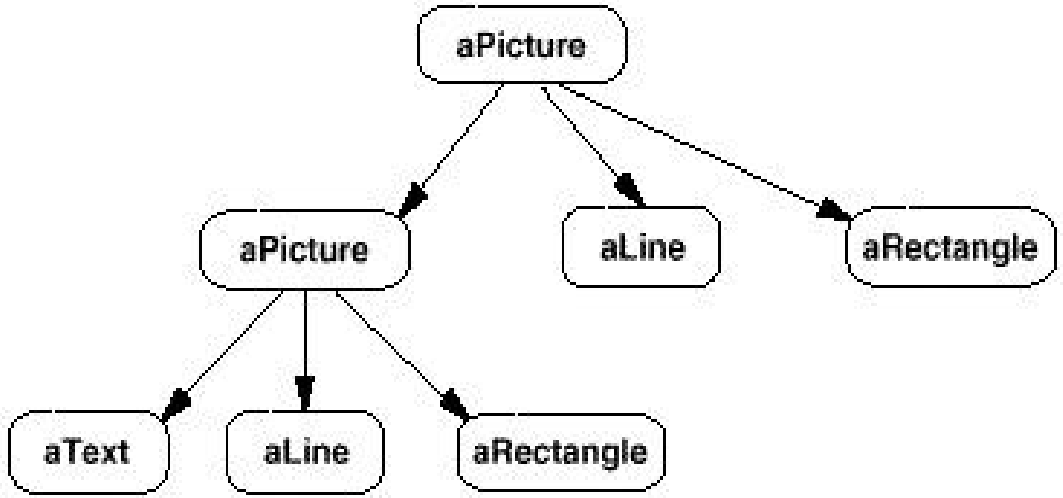
Мотивация

✓ Пример: в графичните редактори потребителя може да построява сложни диаграми, съставени от прости компоненти.



Мотивация

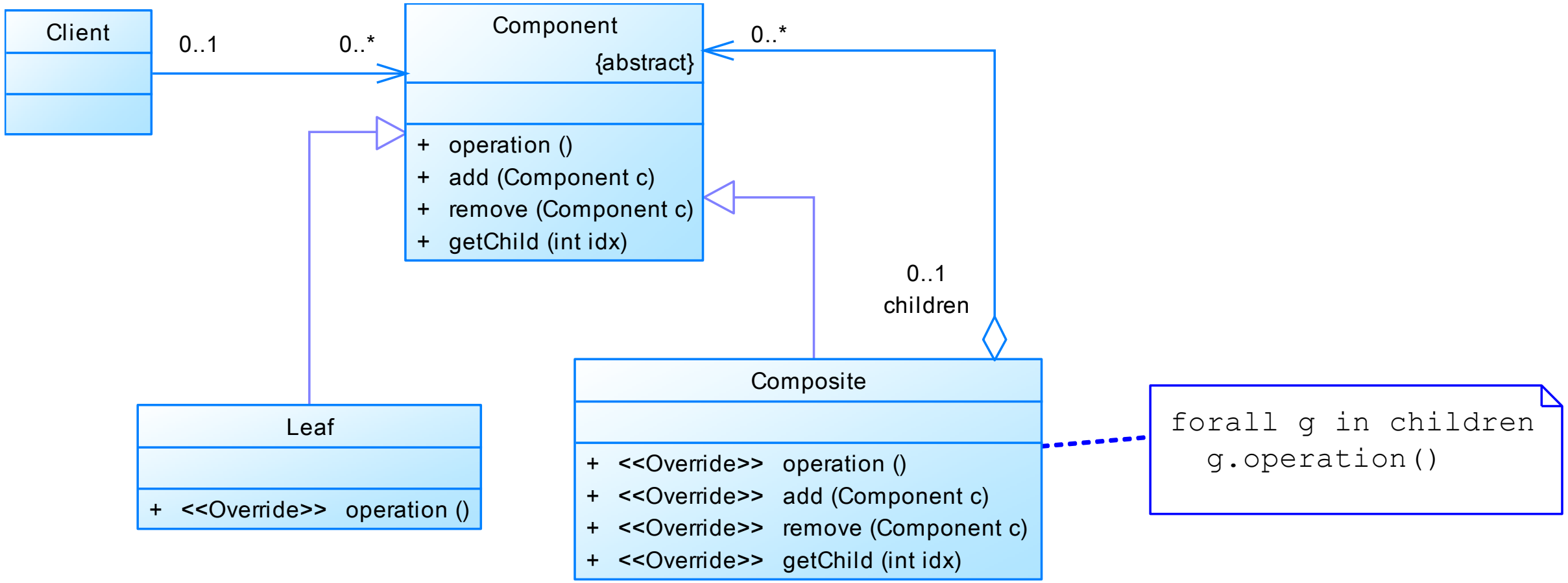
- ✓ Структура на типична композиция, съставена от рекурсивно композирани *Graphic* обекти.



Приложимост

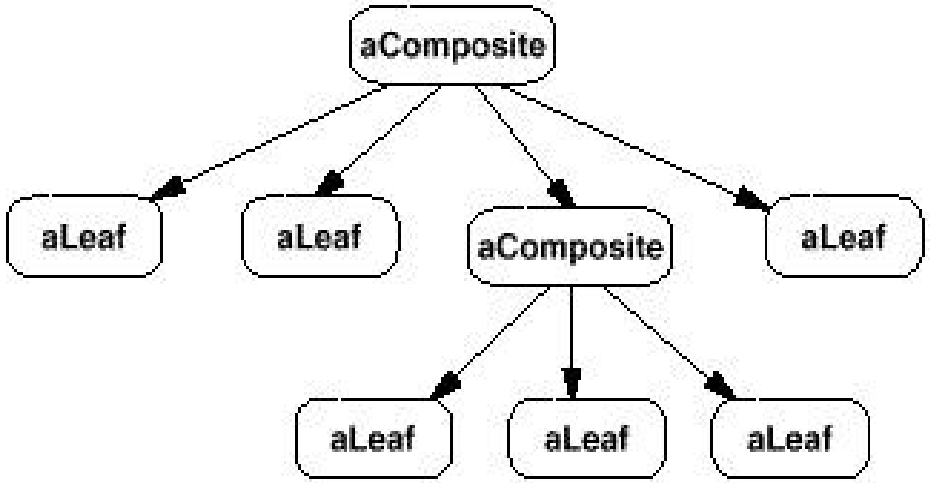
- ✓ **Приложимост:** Шаблонът *Композиция* се използва в следните случаи:
  - За представяне на йерархии от съставни компоненти
  - За да могат клиентите да игнорират разликата между отделни обекти и композиции от обекти. Клиентите ще третират еднакво всички обекти в сложната структура.

Структура


Client
0..1
0..*
Component
{abstract}
+ operation ()
+ add (Component c)
+ remove (Component c)
+ getChild (int idx)
0..*
0..1
children
Leaf
+ <<Override>> operation ()
Composite
+ <<Override>> operation ()
+ <<Override>> add (Component c)
+ <<Override>> remove (Component c)
+ <<Override>> getChild (int idx)
forall g in children
g.operation()

Структура

- ✓ Типичната структура на обект композиция може да изглежда така:



Участници

- ✓ **Component** (Graphic)
  - Декларира интерфейса на обектите в композицията.
  - Имплементира частите от интерфейса, общи за всички класове.
  - Декларира интерфейс за достъп и управление на вложени компоненти.
  - Незадължително може да дефинира интерфейс за достъп до родителя в рекурсивната структура и го имплементира, ако се налага
- ✓ **Leaf** (Rectangle, Line, Text, …)
  - Представлява обекти листа в композицията. Листото няма наследници.
  - Дефинира поведение за примитивните обекти в композицията.
- ✓ **Composite** (Picture)
  - Дефинира поведение за компоненти с наследници.
  - Съхранява наследници.
  - Имплементира операции, свързани с наследници, в интерфейса на *Component.*
- ✓ **Client** – манипулира обекти от композицията чрез интерфейса на *Component.*

Взаимодействия

- ✓ Клиентите използват интерфейса на класа *Component,* за да взаимодействат с обекти на композицията.
- ✓ Ако обектът е листо, заявката се обработва направо.
- ✓ Ако обектът е композиция обикновено заявката се препредава към компонентите наследници, евентуално извършвайки допълнителни операции преди и/или след предаването.

Следствия

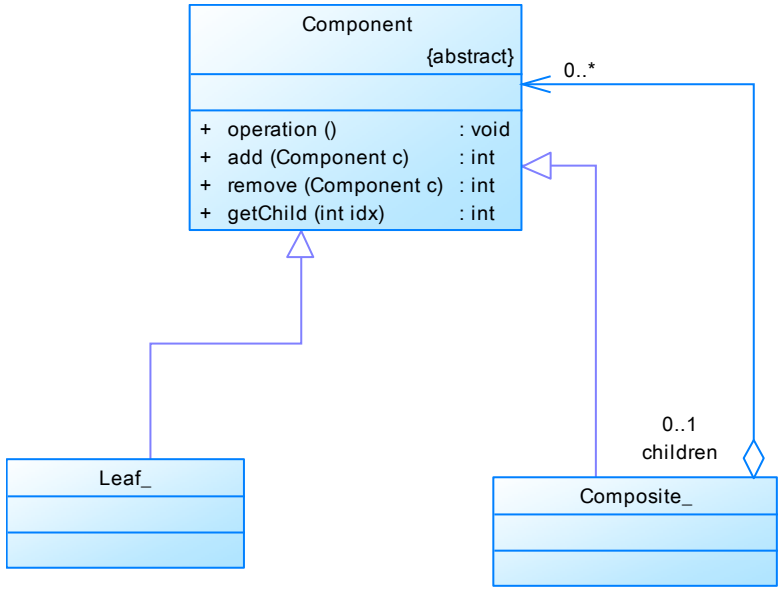
- ✓ **Предимства**
  - Лесно се добавят нови видове компоненти
  - Опростява клиента – не е нужно да знае дали работи с листо или с обект композиция
- ✓ **Недостатъци**
  - Трудно може да се направи ограничение на типовете обекти, които могат да влизат в състава на композиция.

Имплементация

- ✓ Обектите композиции знаят кои компоненти съдържат – т.е. децата им. Трябва ли да поддържат и референция към родителя?
  - Опростява се обхождането и управлението на композицията.
  - Опростява движението нагоре по структурата и изтриването на компонент.

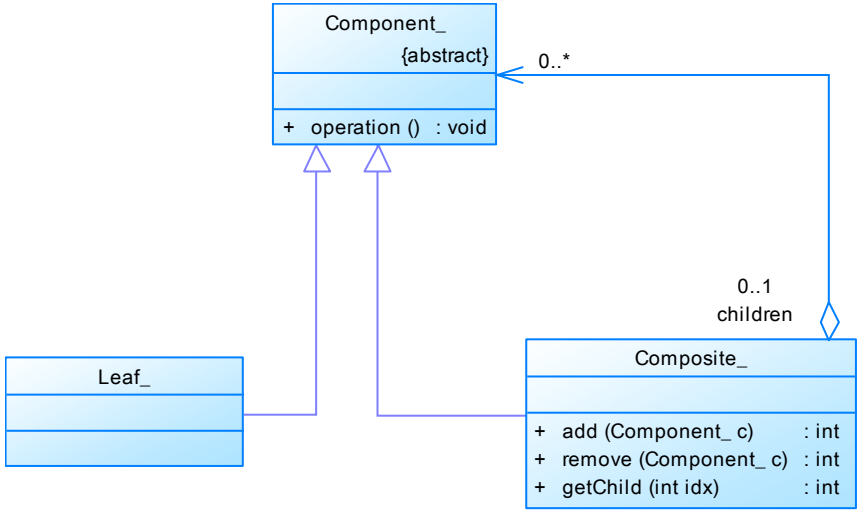
Имплементация

- ✓ Къде трябва да бъдат декларирани методите за управление на наследниците (*add(), remove(), getChild())?
  - В класа *Component:*
    - Прозрачност – всички компоненти се третират еднакво.
    - Добавя несигурност – клиентите могат да опитат да извършат безсмислени неща.



Имплементация

- ✓ Къде трябва да бъдат декларирани методите за управление на наследниците (*add(), remove(), getChild())?
  - В класа *Composite*:
    - Дава сигурност – компилатора ще хване опити за неправилни операции върху *Leaf* обекти.
    - Губи се прозрачност – *Leaf* и *Composite* компонентите ще имат различен интерфейс



Имплементация

- ✓ Кой трябва да изтрива компонентите
  - В езици без събиране на боклука – най-добре всеки *Composite* да изтрива обектите си при унищожаването му.
  - В Java – не е проблем. Защо?
- ✓ Коя е най-добрата структура за съхраняване на компоненти?
  - Дървета, масиви, хеш таблици, свързани списъци?

Пример 1

- ✓ **Ситуация:** Система с GUI има обекти прозорци (window), които могат да съдържат различни графични обекти (widgets), като: бутони и текстови полета. Прозореца може да съдържа widgets контейнери, които да съдържат други widgets.
- ✓ Решение 1: Проектираме всички widgets с различни интерфейси за изчертаване върху екрана. Трябва да направим метод *update()* в класа *Window*, по следния начин:

```
public class Window {

  Button[] buttons;
  Menu[] menus;
  TextArea[] textAreas;
  WidgetContainer[] containers;
```

Пример 1

```
public void update() {
  if (buttons != null)
    for (int k = 0; k < buttons.length; k++)
      buttons[k].draw();
  if (menus != null)
    for (int k = 0; k < menus.length; k++)
      menus[k].refresh();
  // Other widgets handled similarly.
  if (containers != null)
    for (int k = 0; k < containers.length; k++ )
      containers[k].updateWidgets();
}
...
}
```

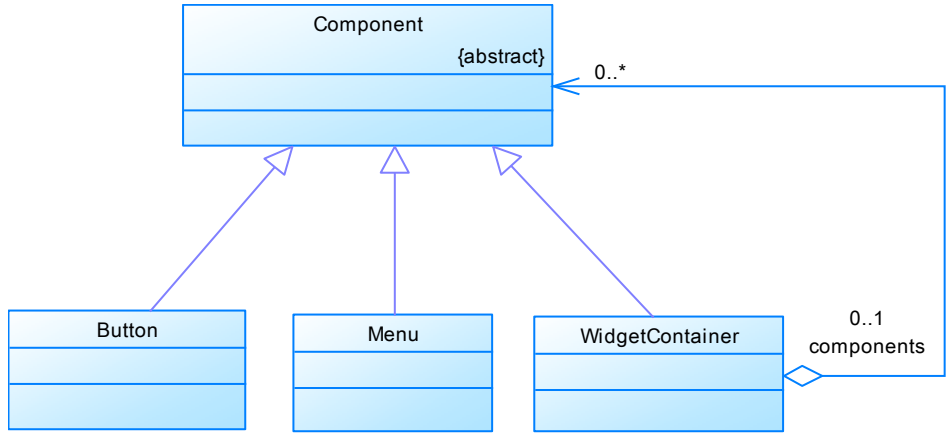
Пример 1

- ✓ Добро ли е това решение?
  - Ако добавим нов компонент – промяна в *update()* метода на *Window.*
- ✓ **Решение 2:** Стремим се да използваме интерфейси. Правим всички widgets да имплементират *Widget* интерфейса – или чрез наследяване на Widget класа или чрез имплементиране на Java Widget интерфейса.

```
public class Window {
  Widget[] widgets;
  WidgetContainer[] containers;
  public void update() {
    if (widgets != null)
      for (int k = 0; k < widgets.length; k++)
        widgets[k].update();
    if (containers != null)
      for (int k = 0; k < containers.length; k++ )
        containers[k].updateWidgets();
  }
}
```

Пример 1

- ✓ Все още правим разлика между widgets и widgets container
- ✓ **Решение 3:** Шаблонът *Композиция*!



Пример 1

- Сега *update()* метода на *Window* изглежда така:

```
public class Window {
  Component[] components;

  public void update() {
    if (components != null)
      for (int k = 0; k < components.length; k++)
        components[k].update();
  }

}
```

Край: Шаблон Композиция

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ