Heavy Object-Oriented Programming in C++

4

edwin.vanouwerkerkmoria@hku.nl

Vorige keer

- **C++ 11 feature**: `auto`

  Laat de compiler uitvogelen wat 't type moet zijn…
  Vooral: declaratie variabelen. Sinds C++ 14 ook return-types van functies

- **SOLID principes**

  single responsibility,open/closed,Liskov,Interface segregation,Dependency reversal

- **Design patterns**

  specifiek: **Strategy**

Opdracht vorige keer

Maak een (tekst-mode) implementatie van Conway's *Game of Life*
https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Vul een matrix (25x25) random met een cel. Voor iedere volgende generatie bereken het aantal buren: Cellen met <2 of >3 buren sterven. Cellen met precies 2 of 3 buren overleven. Lege cellen met 3 buren krijgen nieuwe cel. Herhaal…

- Maak classes voor de verschillende entiteiten
- Implementeer de standaardregels (hierboven) volgens 't strategy-pattern.
- Maak 2 alternatieve implementaties van deze strategy

Vandaag

Templates!

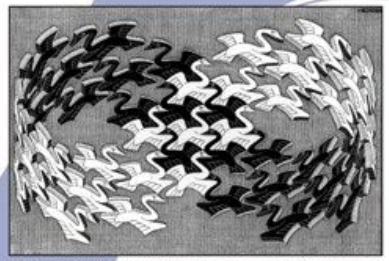
generic programming in C++

Design pattern: **Decorator**

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

DESIGN PATTERNS

Explained **simply**

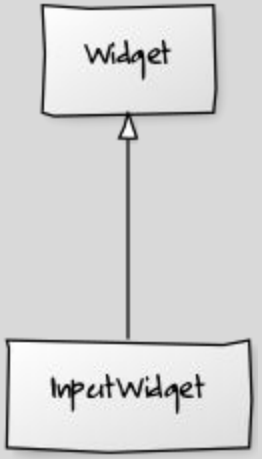

Closet
3,76
Bath. 1
3,80
Nook
GuestRoom1
GuestRoom2
BedRoom2
BedRoom1
Storage
Laundry
Dressing
Porch
Sun Deck
10FEET
3m

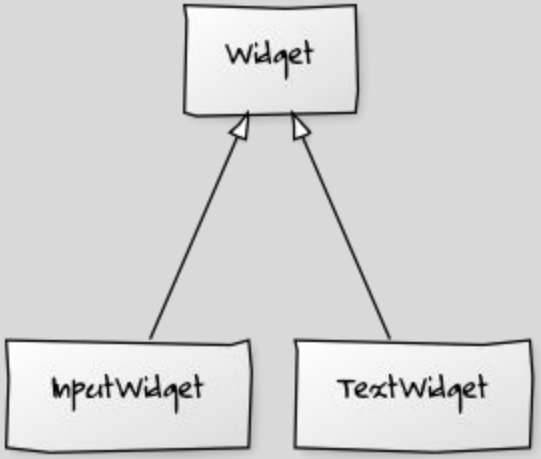
Design pattern: *Decorator*

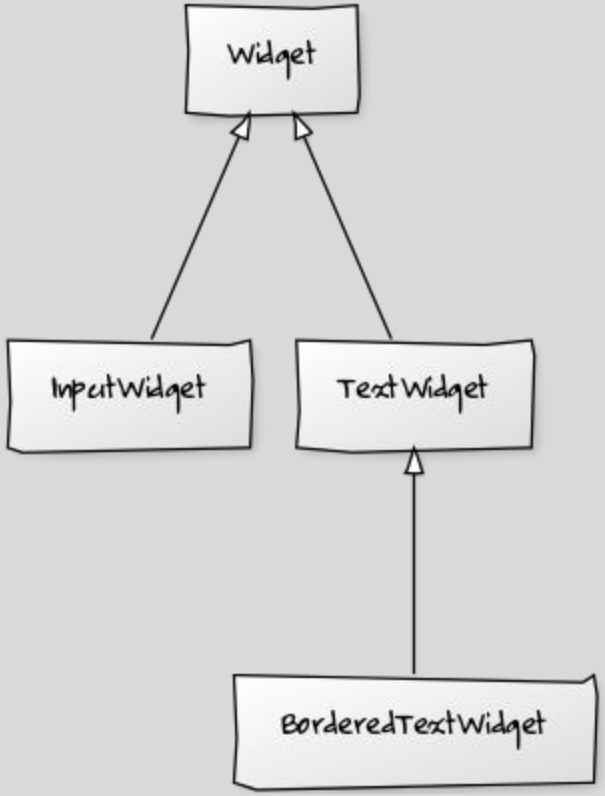
Koppel uitwisselbaar gedrag los van inheritance: voorkom diepe inheritance-hierarchie, voorkom code-duplicatie

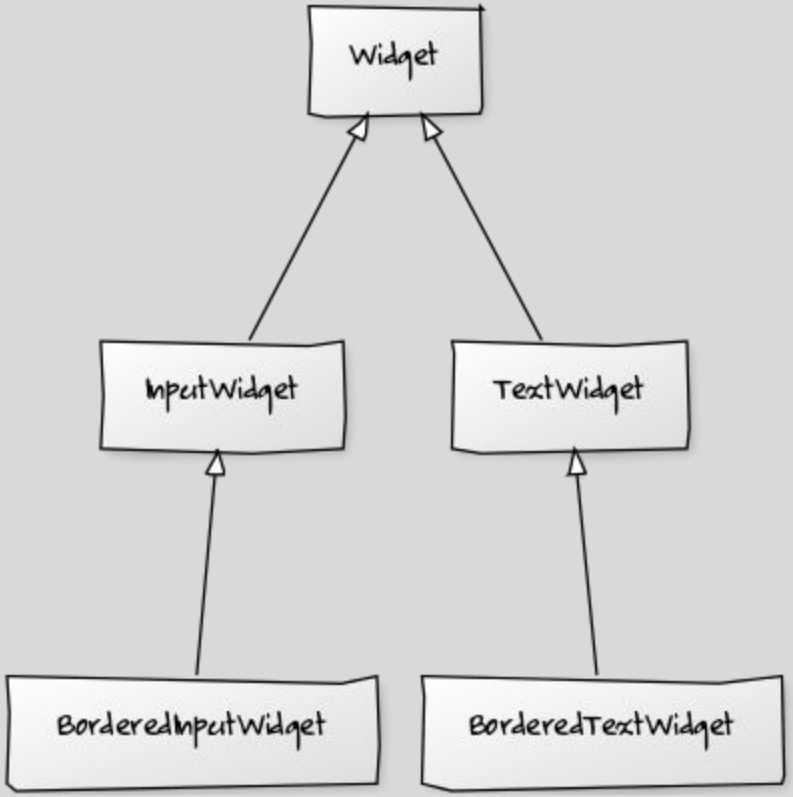
- Maak een baseclass
- Maak een tweede baseclass voor het uitwisselbare gedrag
- Maak subclasses voor specifieke versies van 't gedrag
- Koppel (runtime of compile-time) gedrag aan gebruiker

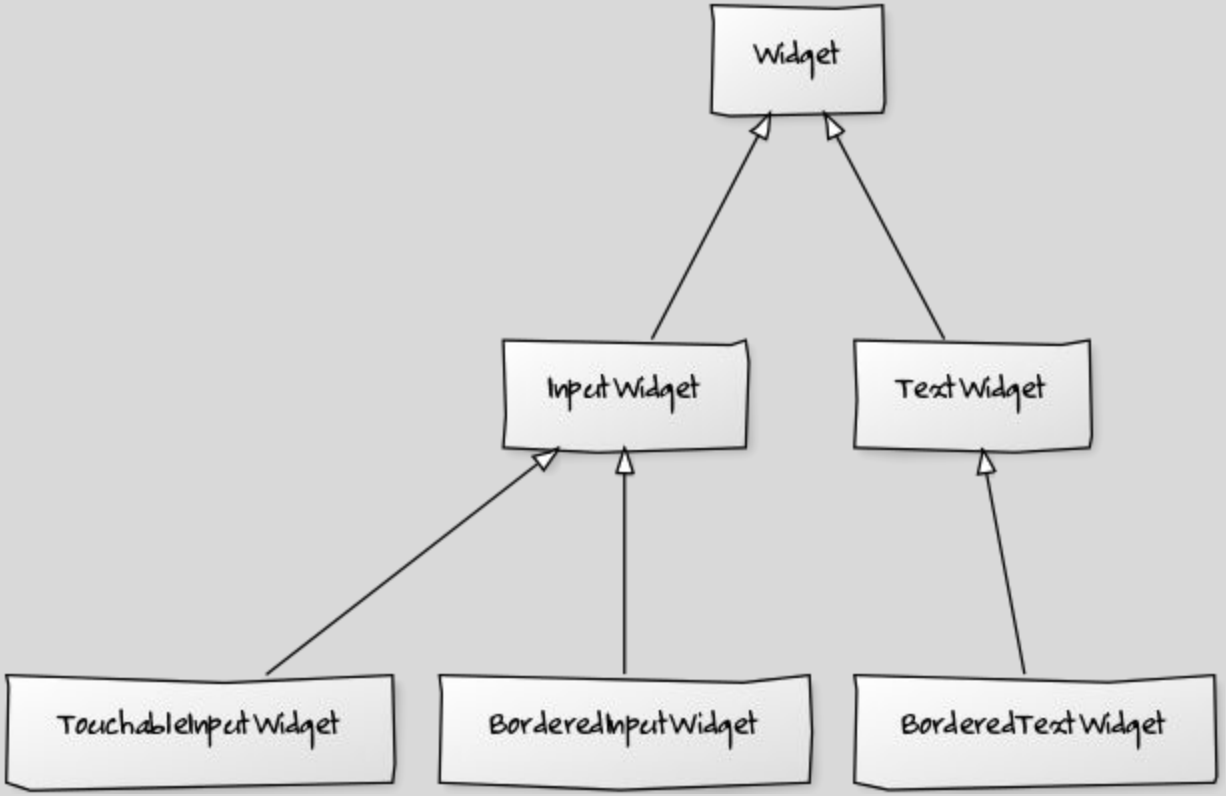

Widget

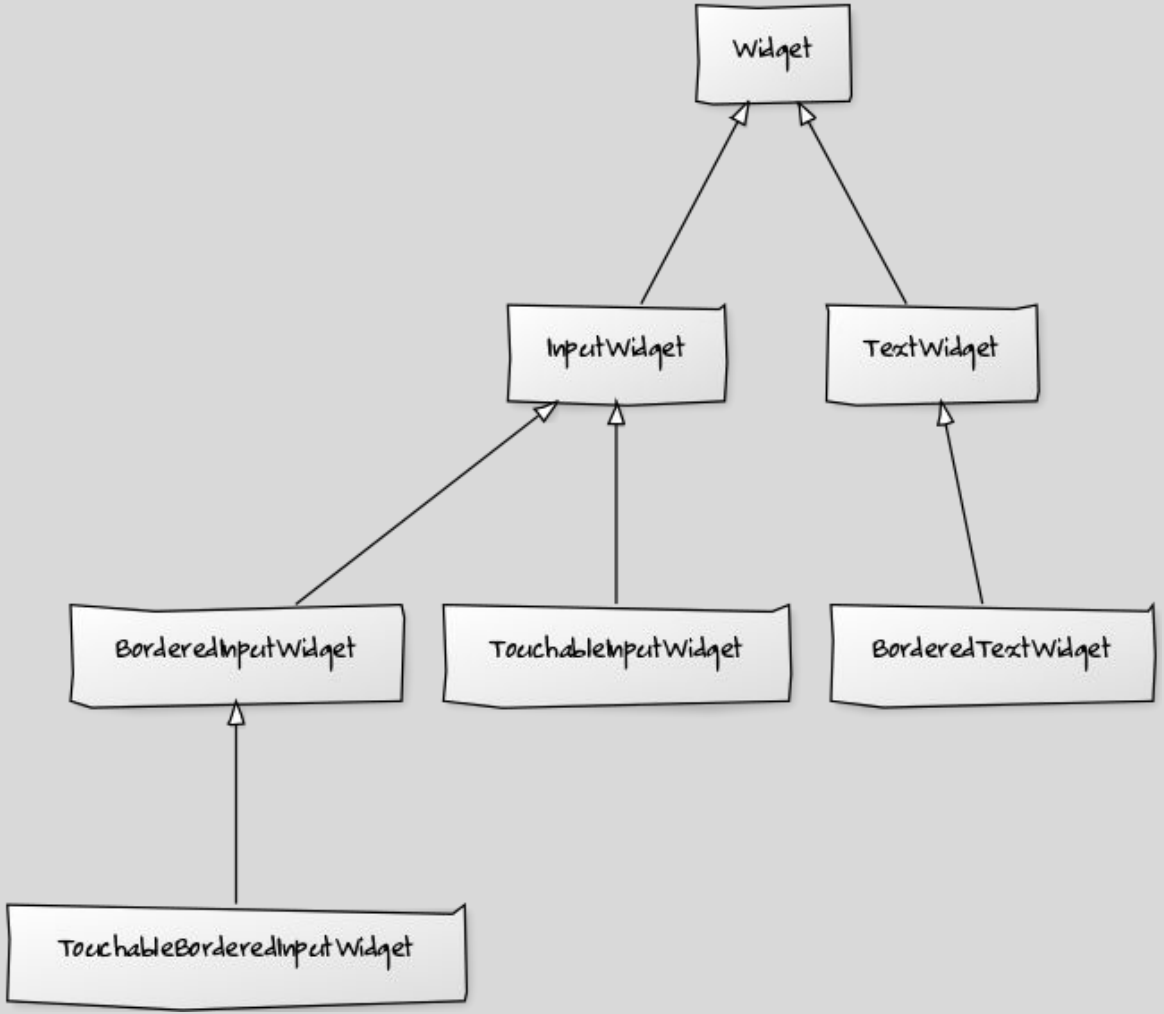

Widget
InputWidget

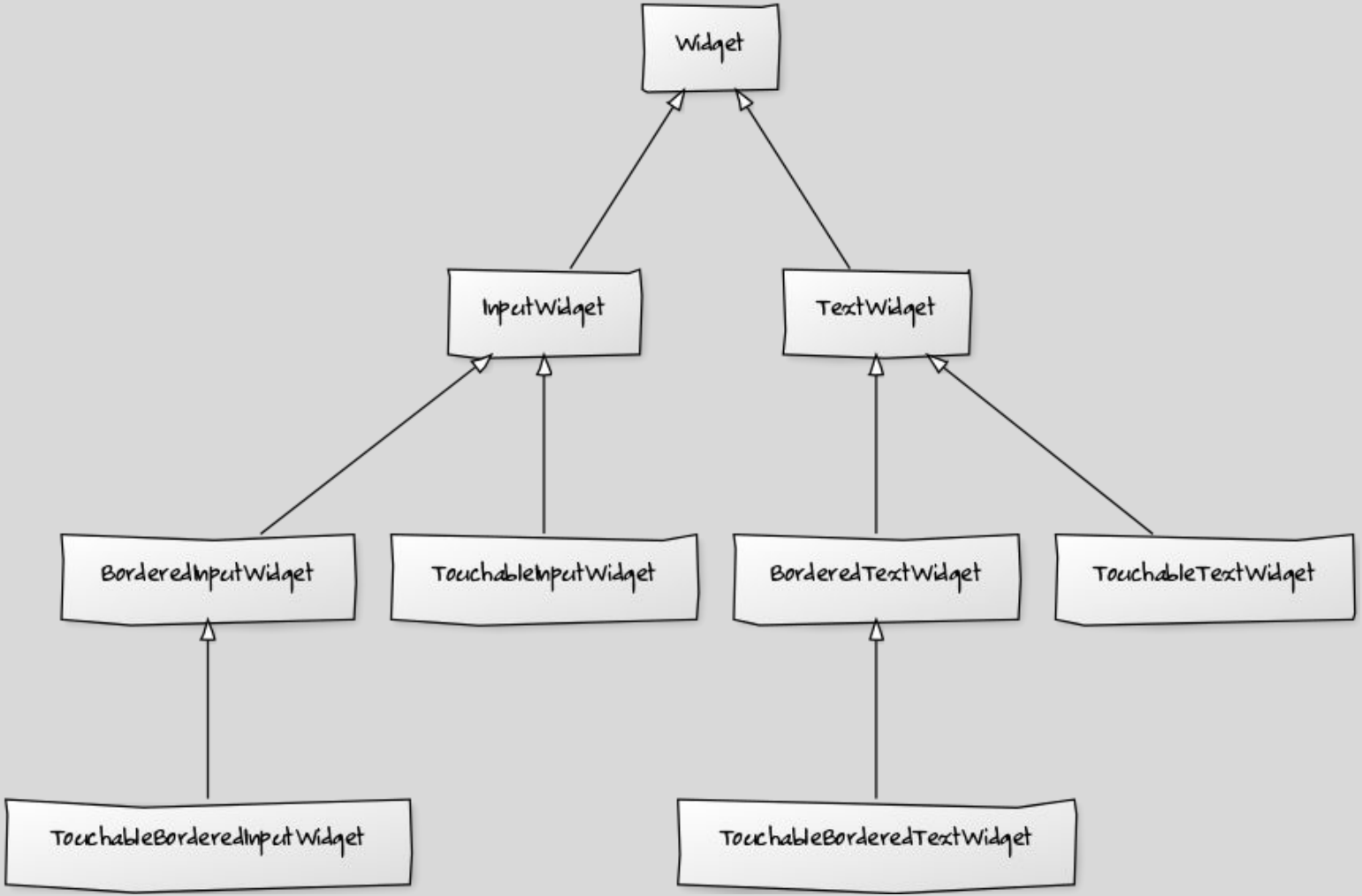

Widget
InputWidget
TextWidget

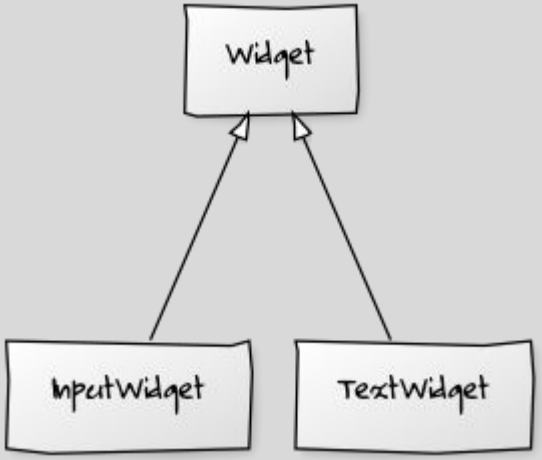

Widget
InputWidget
TextWidget
BorderedTextWidget

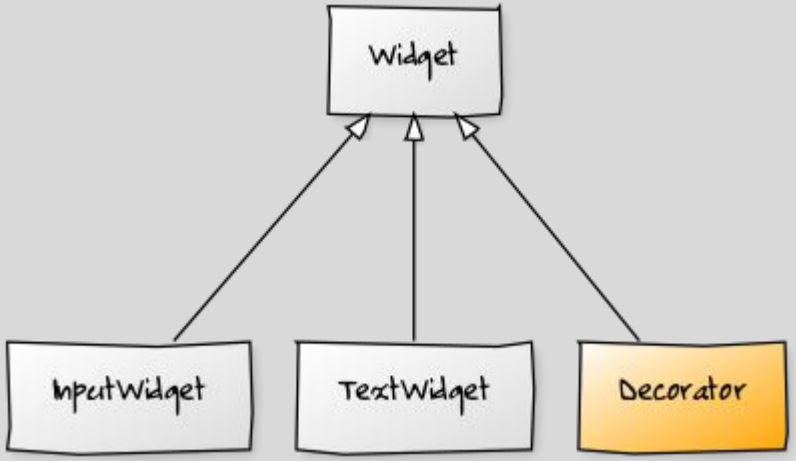

Widget
InputWidget
TextWidget
BorderedInputWidget
BorderedTextWidget

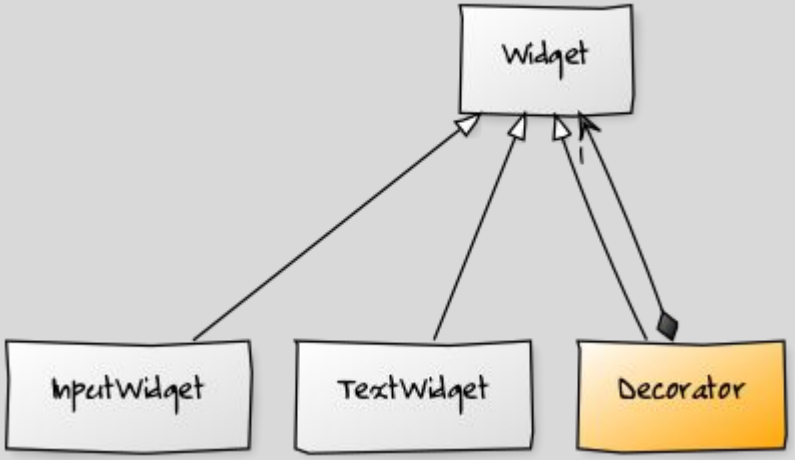

Widget
InputWidget
TextWidget
TouchableInputWidget
BorderedInputWidget
BorderedTextWidget

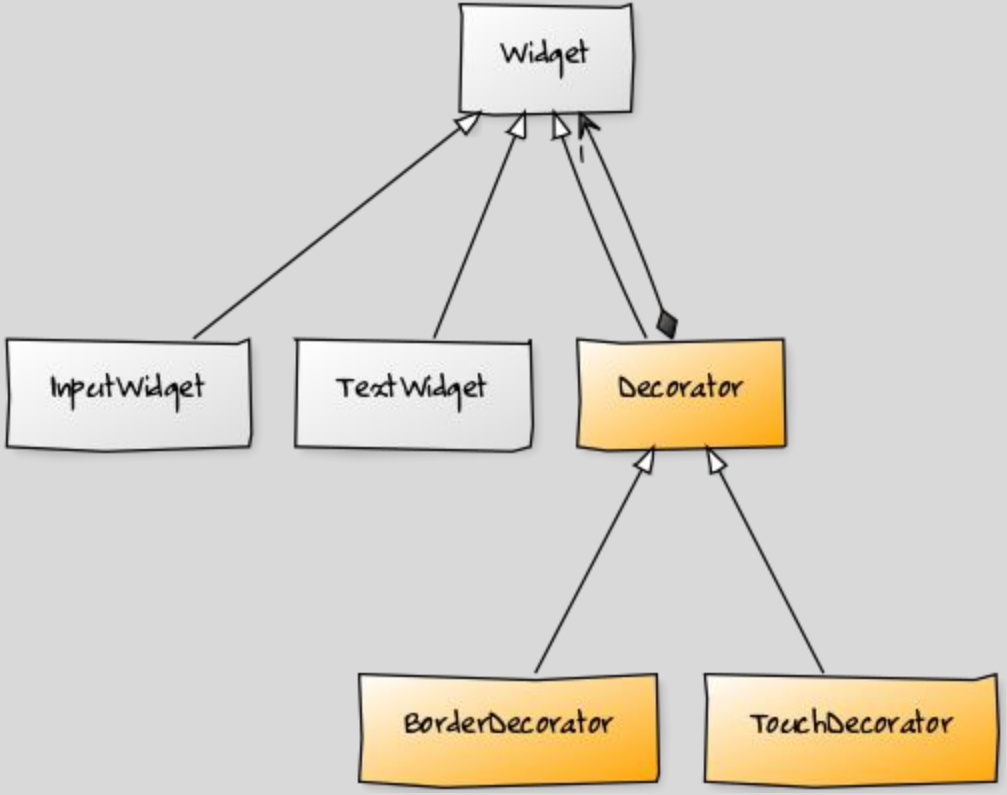

Widget
InputWidget
TextWidget
BorderedInputWidget
TouchableInputWidget
BorderedTextWidget
TouchableBorderedInputWidget


Widget
InputWidget
TextWidget
BorderedInputWidget
TouchableInputWidget
BorderedTextWidget
TouchableTextWidget
TouchableBorderedInputWidget
TouchableBorderedTextWidget


Widget
InputWidget
TextWidget


Widget
InputWidget
TextWidget
Decorator


Widget
InputWidget
TextWidget
Decorator


Widget
InputWidget
TextWidget
Decorator
BorderDecorator
TouchDecorator


GEOMETRIC
ABSTRACT
STYLE
LOREM IPSUM


TEXT HERE


FLYER COVER
HEADLINE
CONSECTET


HEADLINE
FOR FLYER
TEXT HERE


ABSTRACT
PREMIUM
DESIGN
HEADLINE


HEAD
LINE
TEXT HERE


FLYER TITLE


HEADLINE
CLEAN
SIMPLE DESIGN

OO: hergebruik van code

Hergebruik kan via

- classes / objecten
- inheritance

Allemaal vaste data-types, met bijbehorend wisselend gedrag

Wat als je nu hetzelfde gedrag wil hergebruiken, maar dan met wisselende datatypes?

Templates!

Gewone functie

```
int functionname(int parameter) {

    // rest van de functie

}
```

Template functions

```
template<typename T> T functionname(T parameter) {
    // rest van de functie
}
```

of

```
template<typename T>
T functionname(T parameter) {
    // rest van de functie
}
```

template-functies met meerdere types

```
template <typename T, typename U>
void doSomething(T first, U second) {
    ...
}
```

maar wat als er *wel* een returntype is?

template-functies met meerdere types

```
template <typename T, typename U>
int multiply(T first, U second) {
    return first * second;
}
```

return type is afhankelijk van template types T & U

template-functies met meerdere types

```
template <typename T, typename U>
T?/U? multiply(T first, U second) {
    return first * second;
}
```

return type is afhankelijk van template types T & U

template-functie met **auto** return type

```
template <typename T, typename U>
auto multiply(T first, U second) {
    return first * second;
}
```

return type is afhankelijk van template types T & U

template-functie met **auto** return type: **decltype**

```
template <typename T, typename U>

auto multiply(T first, U second) -> decltype( first*second){

          return first * second;

}
```

*return type is afhankelijk van template types T & U: **decltype** verteld de compiler wat er voor **auto** gebruikt moet worden*

Template classes

Zelfde idee als voor een template functie, maar nu voor een hele class

Veel gebruikt bij container classes: `vector<string>`

```
template <typename T>
class MyClass {
    T someMethod(const T& someType);
}
```

Voorbeeld: ContainerShip

Template gotcha's

- voor iedere template instantie, nieuwe class

  meer code om te builden

- template definiëren in *header* file

  veel templates: langere compile

- warrige foutmeldingen van compiler

  VStudio: I'm looking at you… :-(

Opdrachten volgende keer (1)

- Maak één template-functie die array's kan sorteren:
  - Een array van **`strings`**
  - Een array van **`floats`**

- Maak een template-class die Queue-semantics implementeert
  - **`put()`** functie: element toevoegen aan *einde*
  - **`get()`** functie: element ophalen van 't *begin*
  - **`size()`** functie: aantal elementen

  print tussendoor de queue om te testen

Opdrachten volgende keer (2)

Voor een spel modelleer je een serie classes die twee soorten non-playing characters gaan implementeren: elves en orcs. Beide classes hebben 1 attribute: name, en 1 methode: `render()`.

Beide typen NPC zijn er in 3 rollen: Farmer, Soldier, Shaman.

In plaats van 6 subclasses te maken, implementeer je deze varianten via het Decorator pattern. De `render()` methode print de naam, en de rol.