Oprogramowanie Systemów Medycznych

Wykład 2

Programowanie obiektowe

- Wszystko jest obiektem
- Program, to zbiór obiektów, które komunikują się ze sobą przesyłając sygnały
- Każdy obiekt jest określonego typu (klasy) – typ obiektu określa w jaki sposób można się z nim komunikować
- Każdy obiekt, może składać się z wielu innych obiektów, przez co możliwy jest dowolny stopień komplikacji

Założenia

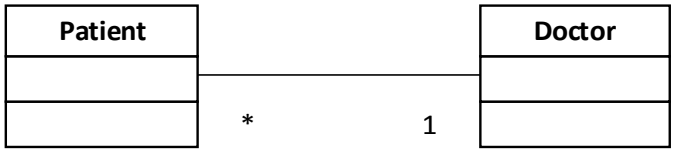
- powiązanie danych i procedur na nich wykonywanych
- ukrywanie implementacji (hermetyzacja)
- wielokrotne używanie tych samych fragmentów programu

Ograniczanie widoczności

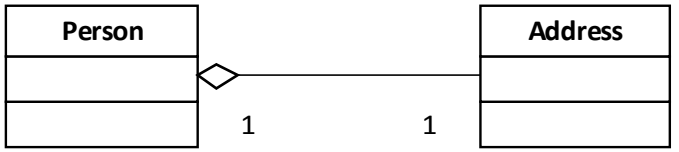
- **private** - pola i metody widoczne tylko o obrębie danej klasy
- (default) - pola i metody widoczne dla wszystkich klas w obrębie tej samej paczki
- **protected** - pola i metody widoczne dla wszystkich klas w obrębie tej samej paczki oraz wszystkich klas potomnych (niezależnie od paczki)
- **public** - dostępne dla wszystkich

Podstawowe relacje między obiektami

- **Asocjacja** – określa zależność między obiektami

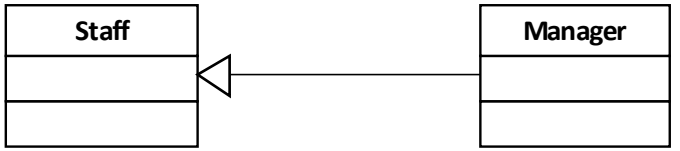


- **Agregacja** – zależność polegająca na tym, że jedna klasa zawiera drugą



- **Kompozycja** – silniejsza niż agregacja zależność, w której obiekt nie tylko zawiera drugą klasę, ale staje się jej wyłącznym właścicielem

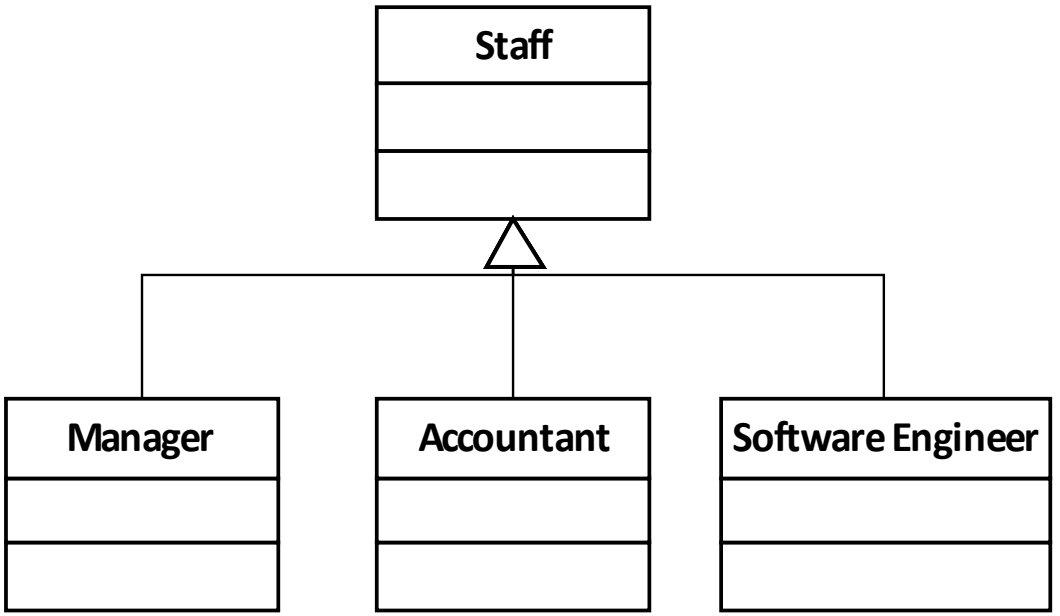
- **Generalizacja/Specjalizacja** - dziedziczenie



Słowa kluczowe

- **`class`** - deklaracja klasy
- **`abstract`** - deklaracja klasy lub metody abstrakcyjnej
- **`extends`** - deklaracja klasy będącej specjalizacją klasy nadrzędnej
- **`implements`** - deklaracja klasy zapewniającej dany interfejs
- **`new`** - tworzenie nowego egzemplarza danej klasy
- **`this`** - odwołanie się do pola lub metody tego egzemplarza klasy
- **`static`** - metoda lub pole klasy jest dostępne bez konieczności tworzenia egzemplarza danej klasy (część wspólna dla wszystkich egzemplarzy)
- **`final`** - pole lub zmienna - przyporządkowana w czasie tworzenia wartość nie może się zmienić; metoda lub klasa - nie może zostać przeładowana przy dziedziczeniu
- **`transient`** - pole nie podlega serializacji
- **`volatile`** - zezwala na modyfikowanie pola przez wiele wątków jednocześnie
- **`synchronized`** - dostęp do metody, lub bloku kodu może uzyskać tylko jeden wątek na raz

Dziedziczenie



Dziedziczenie

- Wszystkie klasy zawsze są potomkami klasy *java.lang.Object*
- Klasa potomna może być specjalizacją tylko jednej klasy (nie ma wielokrotnego dziedziczenia)
- Dziedziczeniu podlegają wszystkie pola oraz metody typu *public*, *protected*, *default*
- Klasa potomna może korzystać ze wszystkich pól oraz metod klasy nadrzędnej w sposób bezpośredni
- Klasy potomne mogą definiować własne metody, których nie posiadają klasy nadrzędne
- Klasy potomne mogą w sposób bezpośredni korzystać z metod klasy nadrzędnej, lub je przeładowywać (polimorfizm)
- Bezpośrednie odwoływanie się do pól lub metod klasy nadrzędnej realizowane jest za pomocą słowa kluczowego **`super`**
- Klasa może zapewniać (implementować) wiele interface'ów

Polimorfizm

- *polimorfizm* w przyrodzie – organizm lub gatunek może przybierać różne formy lub stadia
- *polimorfizm* w programowaniu obiektowym – rodzina klas, może współdzielić pewne funkcje, ale równocześnie klasy potomne mogą definiować własne własne zachowania dla tych samych metod

```
class Dog {
    void sit() {
        //...
    }
    String giveVoice() {
        return "hau";
    }
}

class BadDog extends Dog {
    String giveVoice() {
        return "WRRR#$&*)$#$";
    }
}
```

Interfejs

- *Interfejs jest p*ewnego rodzaju obietnicą w jaki sposób dany obiekt będzie się zachowywał. Interfejs mówi co można zrobić z obiektem, ale nie definiuje jak.
- *Interface* – Jest to typ, który może zawierać jedynie:
  - stałe
  - deklaracje metod
  - typy zagnieżdzone (nie polecam)

  Interfejsy nie mogą być powoływane do życia jako instancje. Mogą jedynie być implementowane, lub rozszerzane.`

```
interface Player {
    public void play();
    public void stop();
}

Player p = new Player(); // ERROR !!!

interface DiskPlayer extends Player {
    public void eject();
}

class DVDPlayer implements DiskPlayer {
    // definicje wszystkich metod interface'u Player i DiskPlayer
}
```

Typy abstrakcyjne

- Klasa abstrakcyjna jest deklarowana poprzez słowo kluczowe *abstract*
- obiekty klasy abstrakcyjnej nie mogą zostać powołane bezpośrednio do życia – tylko klasy specjalizujące
- może zawierać deklaracje metod abstrakcyjnych

```
abstract class Shape {
    private int x, y;

    void setPosition( int x, int y ) {
        this.x = x;
        this.y = y;
    }

    abstract void draw();
}

class Circle extends Shape {
    void draw() {
        // draw cicrcle
    }
}
```

Powoływanie obiektów do życia

```
class Circle extends Shape {
    public Circle() {
        super();
    }
    public Circle( int x, int y ) {
        this();
        this.x = x;
        this.y = y;
    }
    public void draw() {
        System.out.printf("Drawing circle at (%d, %d)\n", x, y);
    }
}
class CircleTest {
    public static void main( String[] args ) {
        Shape a = new Cricle();
        Shape b = new Circle(1, 2);
        // klasa anonimowa
        Shape c = new Shape() {
                public void draw() {
                System.out.printf("Drawing custom shape at (%d, %d)\n", x, y);
            }
        };
        a.draw();
        b.draw();
        c.draw();
    }
}
```

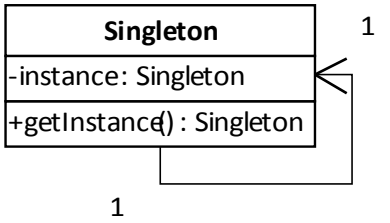
Wzorce Projektowe

- Gang Of Four (GoF) 1994 „*Design Patterns: Elements of Reusable Object-Oriented Software*”
- schematy, które pozwalają na rozwiązywanie częstych problemów, są uniwersalne
- są opisem komunikacji pomiędzy obiektami i klasami, w przypadku uogólnionego problemu, który można zastosować w szczególnym kontekście
- ułatwiają projektowanie złożonych systemów
- są podstawowym narzędziem i standardem przemysłowym – dlatego warto je znać!

Wzorce Projektowe

- **kreacyjne** – pomagające tworzyć nowe obiekty
  - (Abstract) Factory, Builder, Prototype, Singleton
- **strukturalne** – pomagające organizować struktury i powiązania pomiędzy obiektami
  - Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy
- **czynnościowe** – pomagające organizować komunikację oraz zadania pomiędzy instancjami współpracujących ze sobą obiektów
  - Chain of responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template method, Visitor

Singleton



- Tylko jedna sztuka
- Globalny dostęp

```
class Config {

    private static final Config instance = new Config();

    private Config() {
        initialize();
    }

    public static final Config getInstance() {
        return instance;
    }

    [...]
}


Config config = Config.getInstance();
File pwd = config.getWorkingDirecotory();
```

Iterator

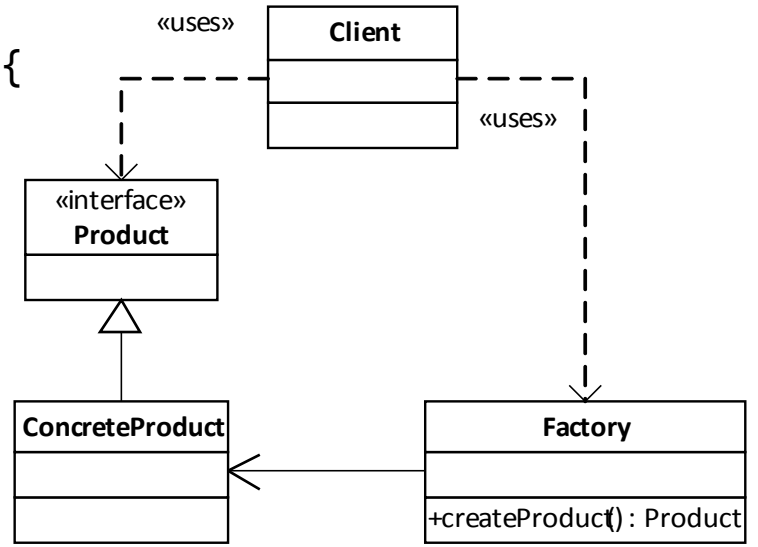
- przechodzenie po kolejnych elementach kolekcji, bez konieczności podawania jej wewnętrznej struktury
- jednolity sposób dostępu do kolejnych elementów

```
    static class MyList implements Iterator<String> {

        String[] list = {"a", "b", "c", "d" };

        int index = 0;

        public boolean hasNext() {
            return index < list.length;
        }

        public String next() {
            return list[ index++ ];
        }

        public void remove() {
        }
    }

    public static void main( String[] args ) {

        Iterator<String> iter = new MyList();

        while ( iter.hasNext() ) {
            System.out.println( iter.next() );
        }
    }
```

Factory

- ukrywanie sposobu tworzenia obiektów
- dostęp do nowo stworzonego obiektu przez wspólny interfejs

```
public class ProductFactory{
    public Product createProduct(String ProductID){
        if (id==ID1)
            return new OneProduct();
        if (id==ID2) return
            return new AnotherProduct();
        ...
        return null;
    }
    ...
}
```



Strategy

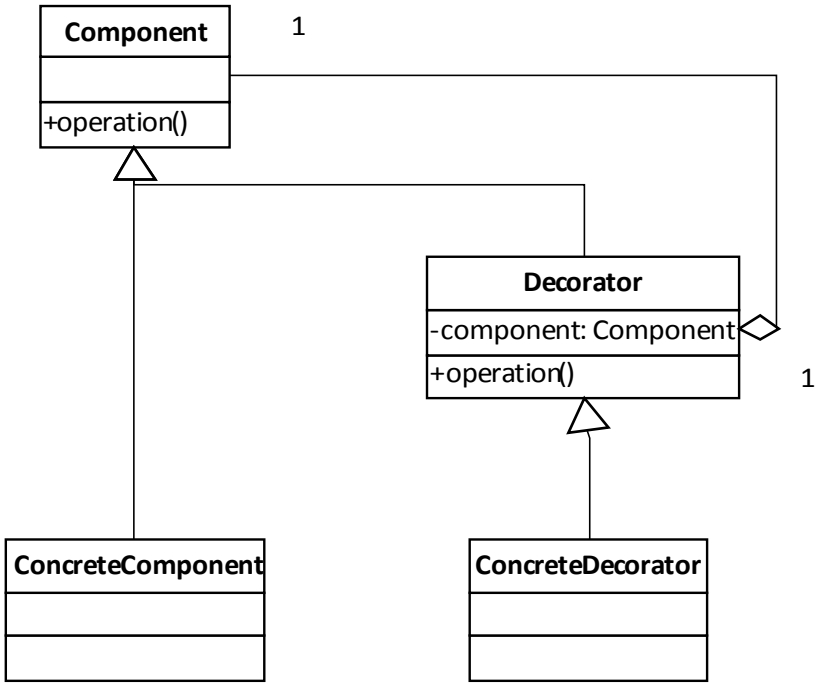
- definiuje wspólny interfejs dla rodziny algorytmów, używanych w zależności od kontekstu

```
interface FindMinima {
    double algorithm( double[] data );
}

class NewthonsMethod implements FindMinima {
    public double algorithm(double[] data) {
        // ...
        return 0;
    }
}
class BisectionMethod implements FindMinima {
    public double algorithm(double[] data) {
        // ...
        return 0;
    }
}

class Solver {
    FindMinima strategy;
    public Solver( FindMinima strategy ){
        this.strategy = strategy;
    }
    double solve( double[] data ) {
        return strategy.algorithm( data );
    }
}


public static void main ( String[] args ) {
    Solver solver = new Solver( new NewthonsMethod() );
    double solution = solver.solve( new double[] { 1.0, 2.0, 1.0, 0.0 } )
}
```

Decorator

- dodawanie właściwości obiektom w sposób dynamiczny

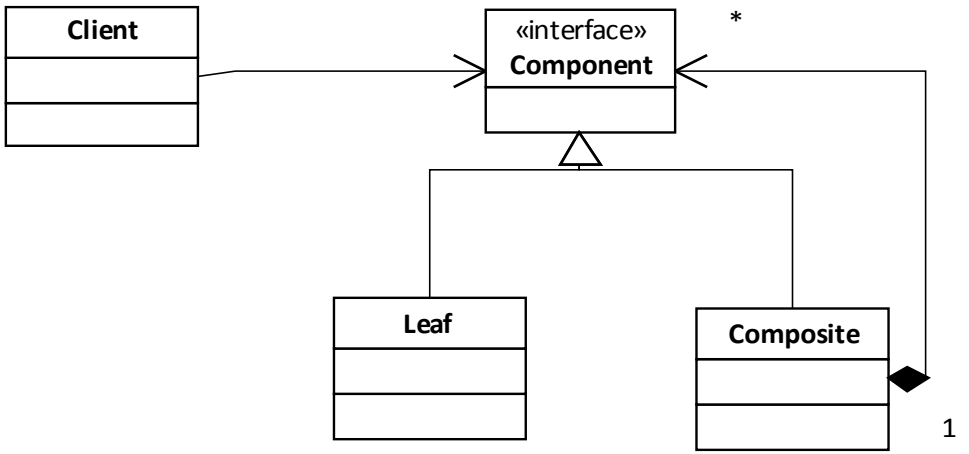
```
interface Component {
    void draw();
}

class Button implements Component {
    public void draw() {
        // draws button
    }
}

abstract class ButtonDecorator implements Component {
    Button decorable;
    public ButtonDecorator( Button button ) {
        this.decorable = button;
    }
}

class NiceFrameButtonDecorator extends ButtonDecorator {
    public NiceFrameButtonDecorator(Button button) {
        super(button);
    }
    void drawNiceFrame() {
        // draws nice frame
    }
    public void draw() {
        decorable.draw();
        drawNiceFrame();
    }
}
```



Composite

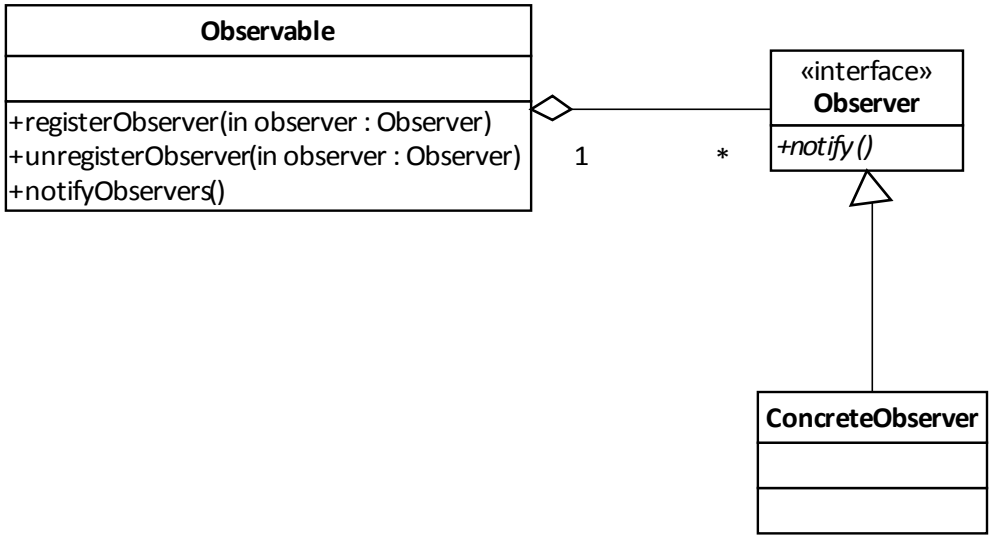
- zagnieżdżanie obiektów w struktury drzewiaste
- klienci kompozytu mogą traktować wszystkie elementy struktury w ten sam sposób, lub całą strukturę jako jeden obiekt

```
interface Component {
    public void paintComponent();
}
class Button implements Component {
    public void paintComponent() {
// painting
    }
}
class Container implements Component {
    Collection<Component> components;
    ...
    public void add( Component component ) {
        components.add( component );
    }
    public void paintComponent() {
        for ( Component comp : components ) {
            component.paintComponent();
        }
    }
}
```

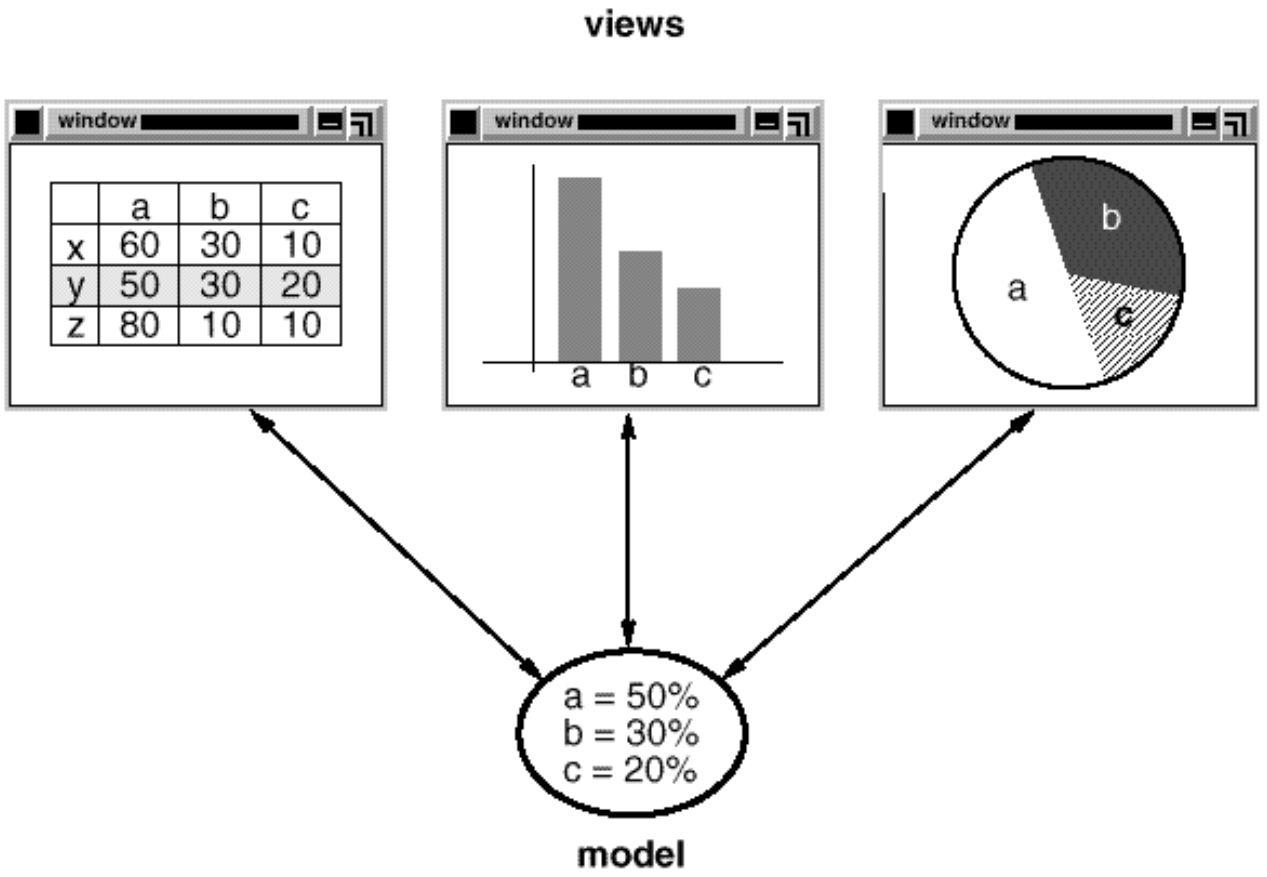


Observer

- "nie dzwoń do nas, my oddzwonimy do Ciebie"
- obiekt informuje obiekty zainteresowane o zmianie swojego stanu

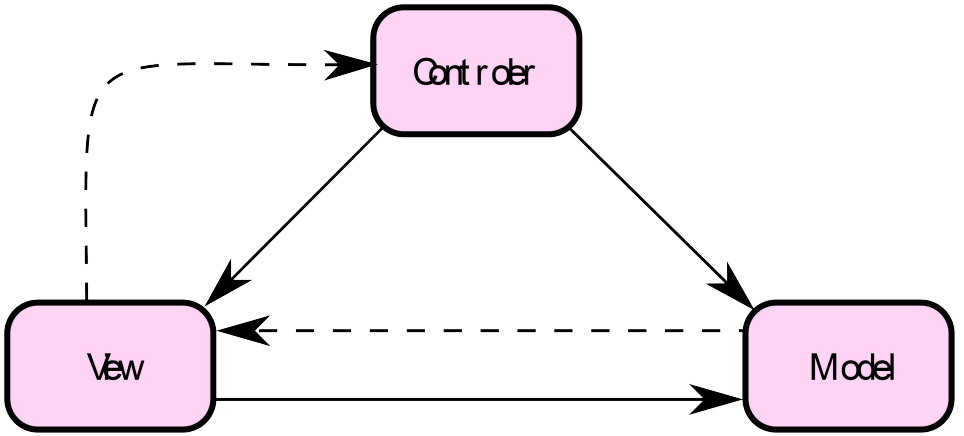


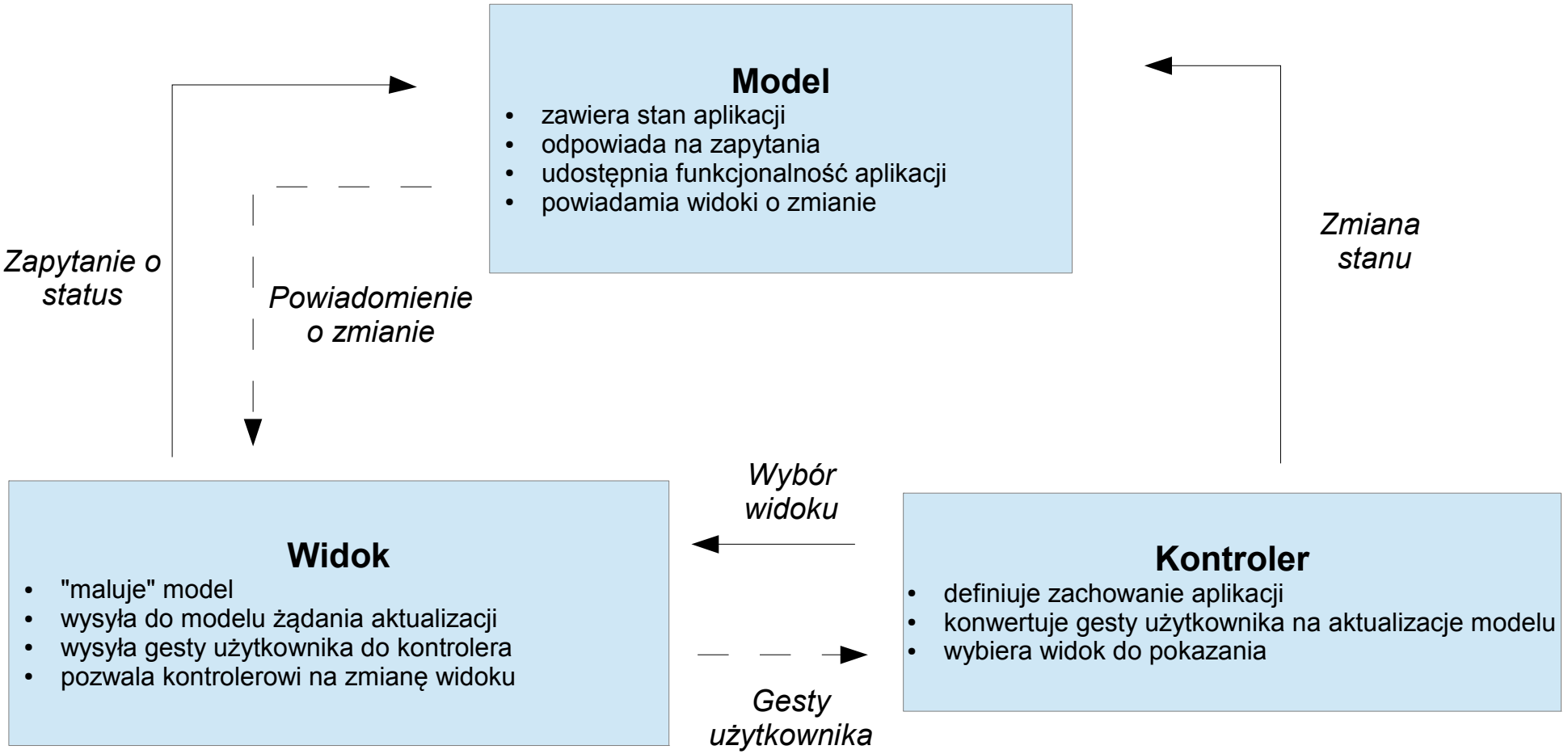
```
interface PropertyChangeListener {
    void propertyChanged( String propertyName, String oldValue, String
newValue )
}

class Model {
    String status = "idle";
    Collection<PropertyChangeListener> observers;

    public Model() {
        observers = new ArrayList<PropertyChangeListener>();
    }
    public void setStatus( String newStatus ) {
        String oldStatus = this.status;
        this.status = newStatus;
        notifyObservers( "status", oldStatus, newStatus );
    }
    public void registerObserver( PropertyChangeListener observer ) {
        observers.add( observer );
    }
    void notifyObservers( String propertyName, String oldStatus, String
newStatus ) {
        for ( PropertyChangeListener observer : observers ) {
            observer.propertyChanged(propertyName, oldValue, newValue);
        }
    }
}


[...]
public static void main( String[] args )
{
    Model model = new Model();
    PropertyChangeListener observer = new PropertyChangeListener() {
        public void propertyChanged(String propertyName, String oldValue,
          String newValue) {
            System.out.println( "property changed" );
        }
    };
    model.registerObserver( observer );

    model.setStatus( "working" );
    model.setStatus( "idle" );
}
```


views
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
model

Model View Controler

- Podstawowy wzorzec projektowania aplikacji z graficznym interfejsem użytkownika. Zapewnia rozdzielenie danych, od sposobu w jaki są wyświetlane i modyfikowane
- **Model** - reprezentuje dane lub pewien problem, zapewnia sposób dostępu do nich
- **View** (widok) - wyświetla graficzną reprezentację danych
- **Controler** - reaguje na interakcję użytkownika, aktualizuje model, podmienia widok




Model
zawiera stan aplikacji
odpowiada na zapytania
udostępnia funkcjonalność aplikacji
powiadamia widoki o zmianie
Zapytanie o status
Powiadomienie o zmianie
Zmiana stanu
Wybór widoku
Widok
"maluje" model
wysyła do modelu żądania aktualizacji
wysyła gesty użytkownika do kontrolera
pozwala kontrolerowi na zmianę widoku
Kontroler
definiuje zachowanie aplikacji
konwertuje gesty użytkownika na aktualizacje modelu
wybiera widok do pokazania
Gesty użytkownika

Typowy schemat działania

- Kontroler jest obserwatorem widoku i modelu. Wszystkie zmiany dokonane na modelu propaguje do widoku i vice-versa
- Widok jest obserwatorem modelu
- Kiedy widok reaguje na gesty użytkownika (np. naciśnięcie guzika, zmianę wartości w polach tekstowych, itd.), powiadamia o tym wszystkich zainteresowanych obserwatorów, w tym zarejestrowany kontroler
- Kontroler modyfikuje model zgodnie ze zmianami, które zostały mu przekazane przez widok
- Model powiadamia swoich obserwatorów o zmianie, w tym kontroler oraz zarejestrowane widoki, w celu ich aktualizacji

```
    static class Model {
        int m = 0;
        private Collection<Listener> observers = new ArrayList<>();
        public void set(int m) {
            this.m = m;
            for ( Listener observer : observers ) {
                observer.modelChanged();
            }
        }
        public void registerObserver( Listener l ) {
            observers.add( l );
        }
    }

    interface Listener {
        void modelChanged();
    }

    interface View {
        void display(Model model);
    }

    static class SimpleView implements View {
        public void display(Model model) {
            System.out.println("m = " + model.m);
        }
    }

    static class MultipledView implements View {
        public void display(Model model) {
            System.out.println("m*m=" + model.m*model.m);
        }
    }

    static class Controler implements Listener {
        private Model model;
        private View  view;
        public Controler( Model model, View view ) {
            this.model = model;
            this.view = view;
            this.model.registerObserver( this );
        }
        public void modelChanged() {
            view.display(model);
        }
    }
```

```
class Test {



    public static void main( String[] args ) {
        Model m = new Model();

        View    simple = new SimpleView();
        View    multiplied = new MultipledView();

        Controler c1 = new Controler(m, simple);
        Controler c2 = new Controler(m, multiplied);

        // interaction
        m.set( 2 );
        m.set( 3 );
    }
}
```