Modelowanie i Analiza Systemów informacyjnych (MAS)

dr inż. Mariusz Trzaska
mtrzaska@pjwstk.edu.pl

Wykład 14

Deklaratywne tworzenie graficznych interfejsów użytkownika

Zagadnienia

- Wstęp
- Klucz do sukcesu
- Założenia proponowanego podejścia deklaratywnego
- Problemy do rozwiązania
- Przykłady użycia
- Podsumowanie

Działający prototyp: https://code.google.com/archive/p/gcl-dsl/

Podejście deklaratywne

- Co to takiego jest?
- Na czym polega różnica w stosunku do klasycznego tworzenia GUI?
- Po co nam kolejne rozwiązanie?
- Zalety
- Wady
- Różne interpretacje
  - deklaratywność komponentowa, np. XAML
  - deklaratywność „modelowa”.

Klucz do sukcesu

- Z jednej strony chcielibyśmy aby komputer stworzył GUI całkowicie (?) samodzielnie – najlepiej bez naszego zaangażowania.
- Równocześnie chcemy aby to GUI było zgodne z naszymi oczekiwaniami dotyczącymi:
  - Funkcjonalności,
  - Estetyki,
  - Użyteczności,
  - Wydajności.

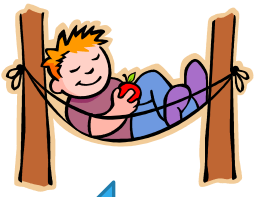
Klucz do sukcesu (2)

- Aby „komputer” mógł to spełnić w 100% musiałby czytać w naszych myślach…
- Jak na razie nie jest to możliwe…
- W związku z tym musimy przekazać mu wskazówki odnośnie tego co ma zrobić.
- Istota podejścia deklaratywnego polega na tym, że mówimy właśnie „**co ma zrobić**”, a nie jak ma to zrobić.

Klucz do sukcesu (3)

- W związku z tym musimy znaleźć:

złoty środek pomiędzy naszym zaangażowaniem w definiowanie GUI, a generycznością rozwiązania oferowanego przez komputer



Słabe dostosowanie do wymagań użytkownika

Doskonałe dostosowanie do wymagań użytkownika

Klucz do sukcesu (4)

- Jeżeli, w celu wygenerowania GUI, system będzie potrzebował zbyt dużej ilości informacji (koniecznych do dostosowania go do szczególnych potrzeb) to korzystanie z niego będzie trudniejsze niż klasyczne stworzenie GUI.
- Wydaje się, że aby zmniejszyć nasz nakład pracy musimy zaakceptować pewną „uniwersalność" rozwiązania, tzn. wygenerowane GUI będą do siebie dość podobne.
- Niemniej, w typowych przypadkach, mogą być bardzo użyteczne.

Założenia proponowanego podejścia deklaratywnego

- Programista określa elementy modelu danych Javy (klasy), które powinny mieć stworzone GUI:
  - Atrybuty,
  - Metody.
- Programista wywołuje metodę, która dla podanego obiektu wyświetli formularz (okno).
- System samodzielnie generuje formularze umożliwiające tworzenie nowych instancji danych, ich edycje, itp.

Założenia proponowanego podejścia deklaratywnego (2)

- Ewentualnie doprecyzowanie szczegółów, m.in.:
  - Etykiety tekstowe,
  - Rodzaje konkretnych widgetów,
- Obsługujemy głównie typy proste (ale nie tylko)
  - Liczby i teksty,
  - Prawda/Fałsz,
  - Datę,
  - Typ wyliczeniowy.
- W celu łatwiejszej implementacji oraz prostszego użycia, rezygnujemy m.in. z:
  - Zaawansowanej walidacji danych,
  - Wielonarodowości (i18n).

Klasyczne rozwiązanie

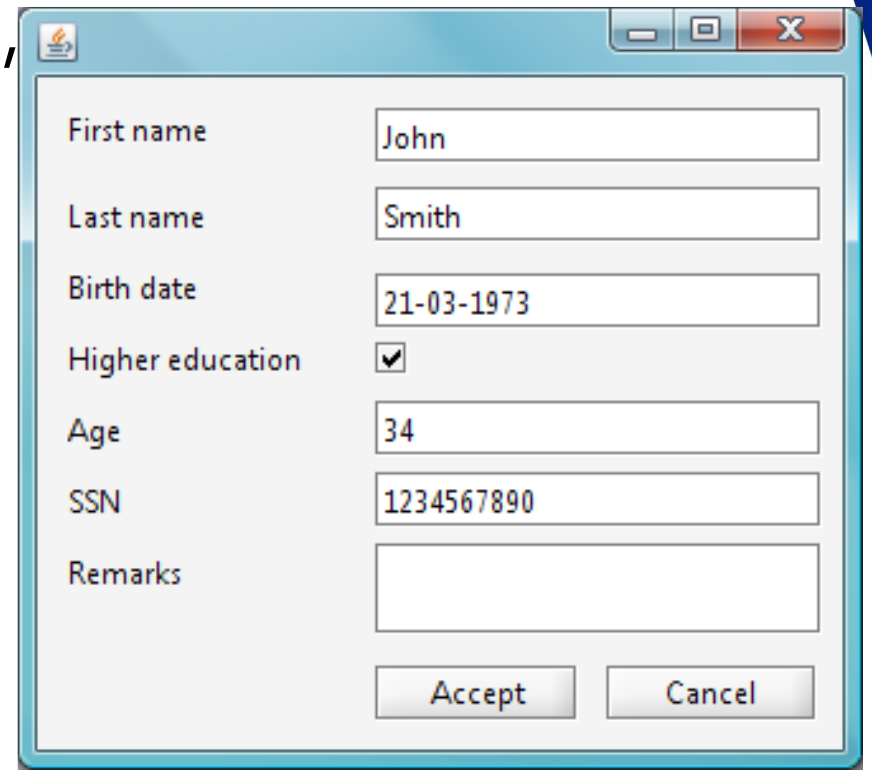
- Aby osiągnąć poprzednią funkcjonalność korzystając z klasycznego podejścia, programista musi:
  a) utworzyć pustą formatkę,
  b) dodać odpowiedni manager rozkładu,
  c) dla każdego atrybutu dodać widget, który zaprezentuje jego zawartość i umożliwi jego edycję,
  d) dla każdej metody dodać widget, który zaprezentuje wynik jej działania,

Klasyczne rozwiązanie (2)

- Aby osiągnąć poprzednią funkcjon. – c.d.
  e) dla każdego widgetu dodać odpowiednią etykietę,
  f) dla każdego widgetu dodać kod, który odczyta wartość określonego atrybutu i wstawi ją do widgetu,
  g) dodać przyciski kontrolne (*Accept*, *Cancel*),
  h) dla przycisku "Accept" dodać kod, który odczyta zawartość widgetów, uaktualni odpowiedni fragment modelu oraz ukryje formatkę,
  i) dla przycisku "Cancel" dodać kod ukrywający.

Klasyczne rozwiązanie (3)

- Realizacja powyższych wymagań oznacza napisanie kilkudziesięciu linii kodu:
  - 7 atrybutów mnożone przez 5 do 10 linii na kontrolkę
  - zarządzanie rozkładem,
  - obsługa przycisków kontrolnych.
- Jigloo GUI Builder wygenerował 105 linii kodu (bez pkt. f, h, i).



Problemy do rozwiązania

- Jak odczytać zawartość klasy (jej strukturę)?
- Jak przekazać informacje, które elementy danych powinny mieć swoje odzwierciedlenie w GUI?
- Jakie widgety powinny być użyte do poszczególnych rodzajów danych?
- Jak połączyć poszczególne elementy GUI (widgety) z danymi (odczyt, zapis)?

Odczyt zawartości klasy

- Wykorzystanie technologii zwanej refleksją (*reflection*).
  - Dostarcza informacje o budowie klas, m.in. atrybuty, metody.
  - Umożliwia
    - tworzenie instancji klas (obiektów),
    - wywoływanie metod,
    - modyfikację, odczyt zawartości atrybutów.

Odczyt zawartości klasy (2)

- Wykorzystanie technologii zwanej refleksją (*reflection*) –c.d.
  - Jest dostępna dla:
    - Javy,
    - C#
    - Częściowo również C++ (RTTI - *Run Time Type Information*).
  - Kluczowa klasa (Java): `Class` i metody:
    - `getDeclaredFields()`,
    - `getDeclaredMethods()`.

Które elementy mają mieć GUI?

- Jak zdefiniować które elementy modelu mają mieć wygenerowane kontrolki GUI?
  - Wszystkie? To się raczej nie sprawdzi bo nie każdy inwariant powinien być dostępny dla użytkownika.
  - Automatycznie? Niestety tak się nie uda dla każdego przypadku.
  - Plik konfiguracyjny, np. XML, *properties*?
  - Przekazywanie odpowiednich parametrów dla metody?
  - Konwencja nad konfiguracją (*Convention over configuration*),
  - Inne możliwości?

Adnotacje klas

- Adnotacje definiowane przez programistę mogą dotyczyć:
  - Klas,
  - Atrybutów,
  - Metod.
- Występują w:
  - Javie,
  - C# (jako *Attributes*).
- Można wykorzystać istniejące lub tworzyć własne.

Adnotacje klas (2)

- Składnia dla Javy

```
@Override
public String toString() {
    return „Moja wlasna implementacja toString()";
}
```

- Składnia dla C#

```
[MyOwnAnnotation]
Public override String ToString() {
    return „Moja wlasna implementacja toString()";
}
```

- Własne adnotacje w Javie

```
public @interface MyOwnAnnotation {
}
```

Adnotacje klas (3)

- Dostosowywanie własnych adnotacji
  - Domyślne parametry,
  - Stosowanie do:
    - Atrybutów,
    - Konstruktorów,
    - Zmiennych lokalnych,
    - Pakietów,
    - Metod,
    - Typów (klasy, interfejsy, typy wyliczeniowe).
  - Dostępność w czasie:
    - kompilacji,
    - działania programu.

Dostosowywanie generowanego GUI

- Do określenia, które elementy modelu mają mieć wygenerowane GUI stworzymy odpowiednie adnotacje.
- Muszą mieć one określone parametry pozwalające na dostosowanie wygenerowanych widgetów.
- Liczba tych parametrów powinna być jak najmniejsza.
- W celu dalszego ułatwienia korzystania ze stworzonych adnotacji, parametry będą miały domyślne wartości.

Dostosowywanie generowanego GUI (2)

- Dzięki wprowadzeniu podziału możemy indywidualnie kształtować funkcjonalność adnotacji dla:
  - Atrybutów,
  - Metod.
- Zestaw parametrów będzie podobny, ale domyślne wartości różne.

Dostosowywanie generowanego GUI (3)

- Parametry adnotacji
  - `label`. Opisuje etykietę znajdującą się przy widgecie. Jeżeli jest pusta (wartość domyślna) to zostanie wykorzystana nazwa atrybutu lub metody. Ten parametr jest wymagany bo czasami musimy wprowadzić specjalne nazewnictwo (np. spację, czy polskie litery);
  - `widgetClass`. Klasa widgetu, która będzie wykorzystana do edycji oraz wyświetlania danych. Domyślna wartość zakłada użycie pola tekstowego (`JTextBox`);

Dostosowywanie generowanego GUI (4)

- Parametry adnotacji – c.d.
  - `tooltip`. Krótki tekst wyświetlany po najechaniu kursorem na element.
  - `getMethod`. Metoda wykorzystywana do odczytu wartości atrybutu. Domyślna wartość zawiera pusty ciąg i oznacza zastosowanie podejścia opartego na getterach oraz setterach;
  - `setMethod`. Analogicznie jak w przypadku `getMethod`, ale dotyczy zapisu wartości;
  - `showInFields`. Flaga określająca, czy ten element powinien być widoczny w formatce z polami;

Dostosowywanie generowanego GUI (5)

- Parametry adnotacji – c.d.
  - `showInTable`. Flaga określająca, czy ten element powinien być widoczny w widoku tabelarycznym;
  - `showInSearch`. Flaga określająca, czy ten element powinien być widoczny w widoku służącym do wyszukiwania;
  - `order`. Liczba określająca kolejność widgetu w formularzu;
  - `readOnly`. Flaga definiująca, czy dane są dostępne w trybie tylko do odczytu (bez możliwości modyfikacji);

Dostosowywanie generowanego GUI (6)

- Parametry adnotacji – c.d.
  - `scaleWidget`. Określa, czy widget powinien zmieniać swój rozmiar w czasie modyfikacji wielkości formularza.

Tworzenie własnych adnotacji

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface GUIGenerateAttribute {
    /**
     * Podaje nazwe  etykiety dla pozycji w GUI. Gdy "" to bedzie uzyta nazwa
    atrybutu (pierwsza litera skonwertowana do duzej).
     * @return
     */
    String label() default "";

    /**
     * Definiuje widget, ktory bedzie stworzony dla obslugi danego atrybutu.
     * Musi dziedziczyc z "java.awt.Component" ("javax.swing.JComponent")
     * W tym elemencie MUSI byc metoda "String getText()" oraz "setText(String)".
     * @return
     */
    String widgetClass() default "javax.swing.JTextField";

    /**
     * Definiuje tooltip wyswietlany dla widgetu.
     * @return
     */
    String tooltip() default "";
    // [...]
```

Tworzenie własnych adnotacji (2)

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface GUIGenerateAttribute {
    // - c.d.
    /**
     * Definiuje metode do pobierania wartosci dla atrybutu. Jezeli jest ""
    oznacza to wykorzystanie metody "String get<NazwaAtrybutu>()"
     * @return
     */
    String getMethod() default "";

    /**
     * Definiuje metodę do ustawiania wartosci dla atrybutu. Jezeli jest ""
    oznacza to wykorzystanie metody "set<NazwaAtrybutu>(String)"
     * @return
     */
    String setMethod() default "";

    /**
     * Definiuje czy generowac widget przy widoku w polach (przegladanie lub
    edycja).
     * @return
     */
    boolean showInFields() default true;
```

Tworzenie własnych adnotacji (3)

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface GUIGenerateAttribute {
    // - c.d.

    /**
     * Definiuje czy generowac widget przy widoku tabeli.
     * @return
     */
    boolean showInTable() default true;

    /**
     * Definiuje czy generowac widget przy widoku w polach do wyszukiwania.
     * @return
     */
    boolean showInSearch() default true;

    /**
     * Okresla kolejnosc wygenerowanegu widgetu wsrod innych wygenerowanych
    widgetow.
     * @return
     */
    int order() default 0;
    // ...
```

Tworzenie własnych adnotacji (4)

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface GUIGenerateAttribute {
    // - c.d.

    /**
     * Gdy True to wartosc atrybutu nie moze byc zmieniona.
     * @return
     */
    boolean readOnly() default false;

    /**
     * Gdy true to w czasie zmian wielkosci formy dostosowuje wielkosc widgetu.
     * @return
     */
    boolean scaleWidget() default false;
}
```

Dobór kontrolek GUI dla danych

- Najprostsze i najbardziej uniwersalne podejście wykorzystuje `JTextBox`,
- Oczywiście można je ulepszyć:
  - Specjalne `ComboBox`'y do pracy z typem wyliczeniowym,
  - `CheckBox`'y obsługujące typ `boolean`,
  - Inne – zdefiniowane przez programistę.

Przesyłanie danych do/z kontrolki

- Ponieważ nasze rozwiązanie jest generyczne (tzn. zdolne do pracy z różnymi typami danych) musimy ustalić wspólny „typ” przesyłanych danych: tekst (`String`).
- Wymagamy aby każda wykorzystywana kontrolka obsługiwała dwie metody:
  - `void setText(String)` - zapis danych do kontrolki,
  - `String getText()` – odczyt danych z kontrolki.

Przesyłanie danych do/z kontrolki (2)

- Większość kontrolek udostępnianych przez Swing'a obsługuje w/w metody.
- Czasami musimy stworzyć prosty *wrapper* (opakowanie) dla istniejącej kontrolki tak, aby pracowała z w/w metodami.
- W nielicznych sytuacjach może wystąpić potrzeba wprowadzenia znaczących modyfikacji lub implementacji własnej kontrolki.

Przykłady użycia – proste adnotacje

- Działający prototyp:

https://code.google.com/archive/p/gcl-dsl/

- Prosta klasa dla, której potrzebujemy GUI

```
public class Person {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private boolean higherEducation;
    private String remarks;
    private int SSN;
    private double annualIncome;

    public int getAge() {
        // [...]
    }
}
```

Przykłady użycia – proste adnotacje (2)

- Wprowadzamy proste adnotacje z domyślnymi wartościami – *c.d.*

```
public class PersonAnnotated {
    @GUIGenerateAttribute
    private String firstName;
    @GUIGenerateAttribute
    private String lastName;
    @GUIGenerateAttribute
    private LocalDate birthDate;
    @GUIGenerateAttribute
    private boolean higherEducation;
    @GUIGenerateAttribute
    private String remarks;
    @GUIGenerateAttribute
    private int SSN;
    @GUIGenerateAttribute
    private double annualIncome;
    @GUIGenerateMethod
    public int getAge() {
        // ...
    }
    // Standard getters/setters methods
}
```

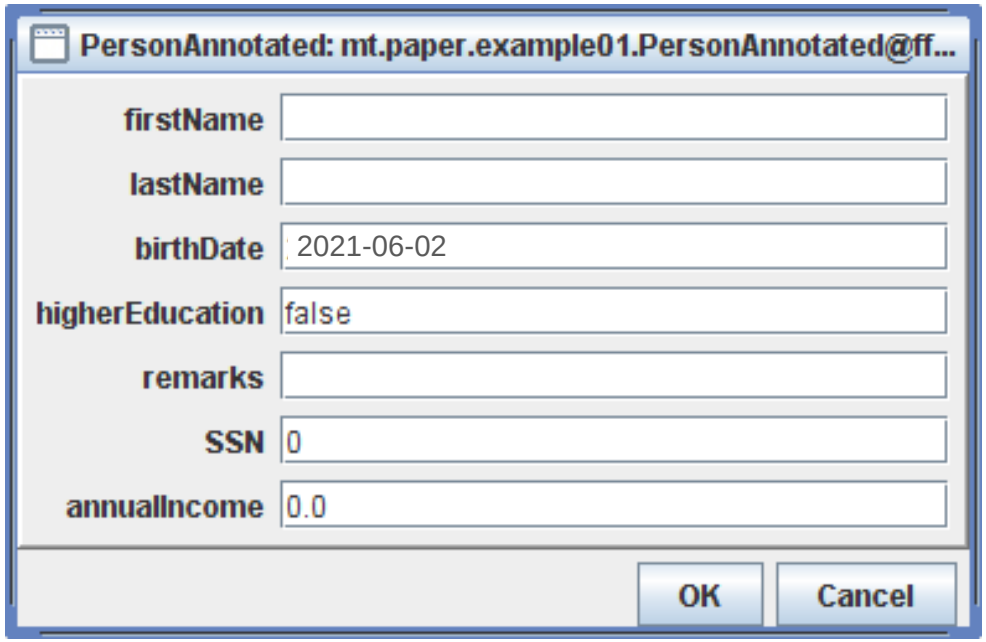
Przykłady użycia – proste adnotacje (3)

- Wprowadzamy proste adnotacje z domyślnymi wartościami – c.d.

```
JDesktopPane jdp = new JDesktopPane();

PersonAnnotated person = new PersonAnnotated();
SingleObjectPlusFrame personFrame = GUIFactory.getObjectFrame(person, jdp,
                                                    false, true, true);
```

- I oto efekt wywołania jednej metody: `GUIFactory.getObjectFrame(.`
- Oczywiście są pewne wady, ale część z nich zaraz usuniemy.



Przykłady użycia – parametryzowane adnotacje

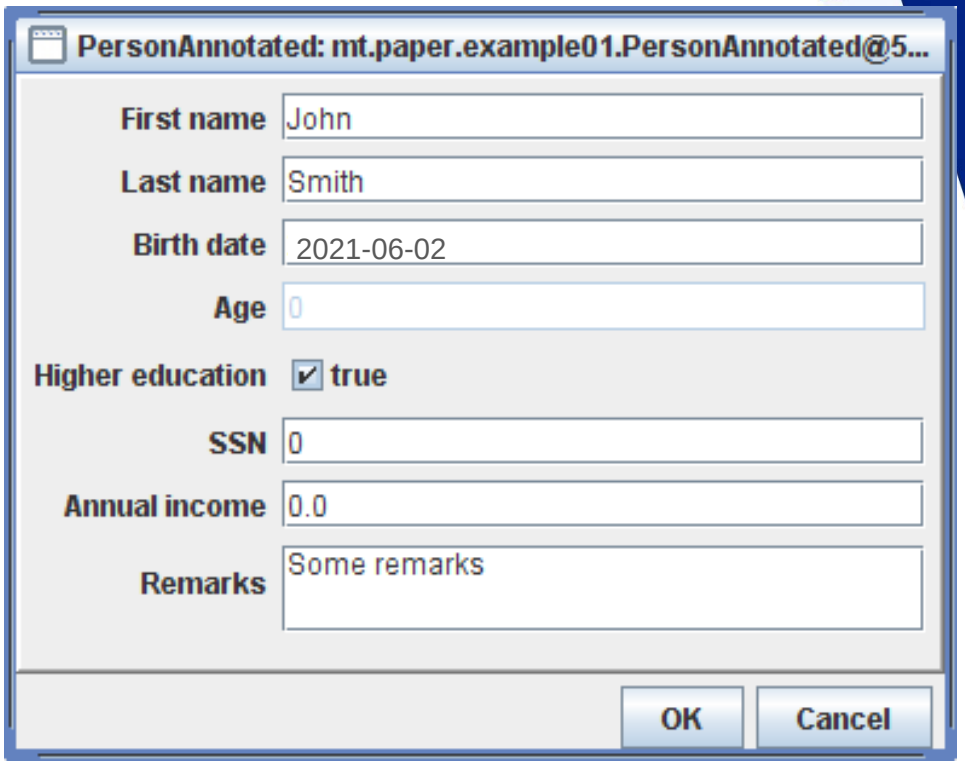
- Tym razem dostosowujemy nasze adnotacje

```
public class PersonAnnotated {
    @GUIGenerateAttribute(label = "First name", order = 1)
    private String firstName;
    @GUIGenerateAttribute(label = "Last name", order = 2)
    private String lastName;
    @GUIGenerateAttribute(label = "Birth date", order = 3)
    private Date birthDate = new Date();
    @GUIGenerateAttribute(label = "Higher education",
    widgetClass="mt.mas.GUI.CheckboxBoolean", order = 5)
    private boolean higherEducation;
    @GUIGenerateAttribute(label = "Remarks", order = 50,
    widgetClass="javax.swing.JTextArea",scaleWidget=false)
    private String remarks;
    @GUIGenerateAttribute(order = 6)
    private int SSN;
    @GUIGenerateAttribute(label= "Annual income", order=7)
    private double annualIncome;
    @GUIGenerateMethod(label="Age", showInFields = true, order = 4)
    public int getAge() { ...}

    // Standard getters/setters methods
}
```

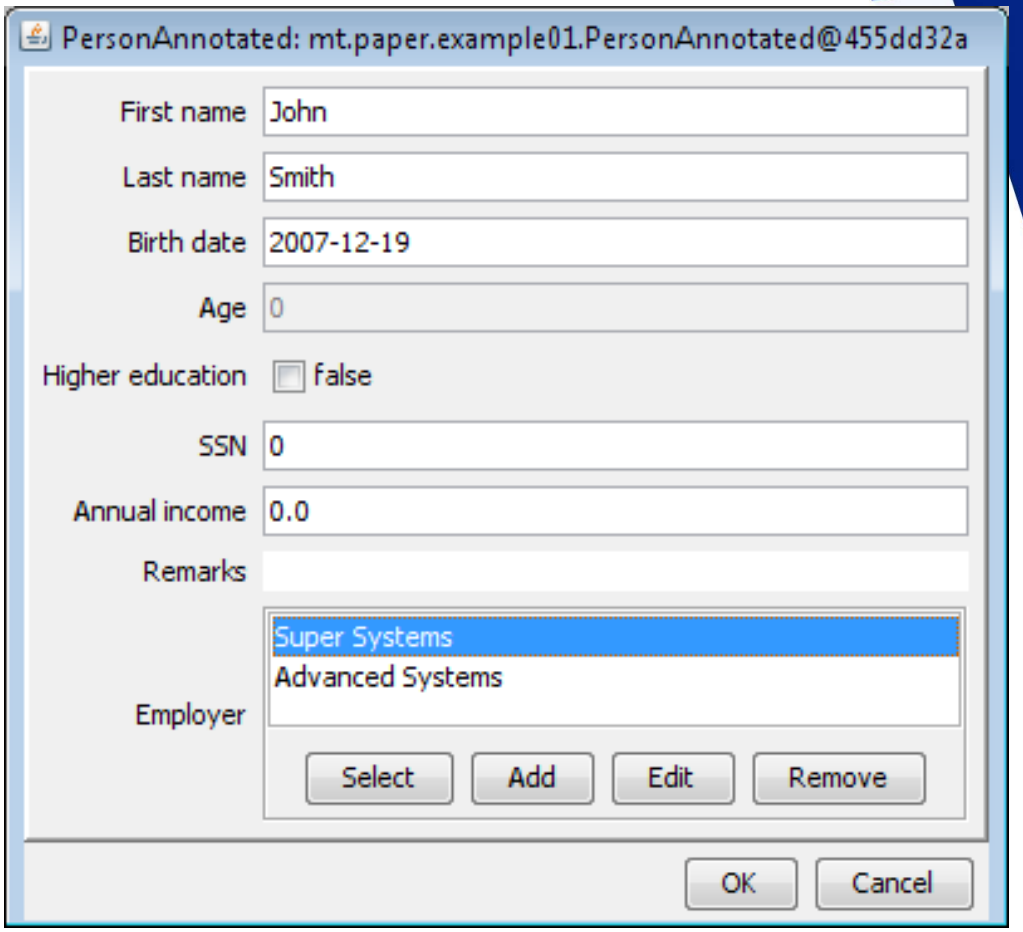
Przykłady użycia – parametryzowane adnotacje (2)

- Wywołanie jak poprzednio, ale dzięki przekazaniu odpowiednich parametrów dla adnotacji:
  - Poprawne nazwy atrybutów,
  - Właściwa kolejność,
  - Wiek jest tylko do odczytu,
  - Pole *Remarks* podlega skalowaniu,
  - Dedykowane kontrolki (`JCheckBox`).



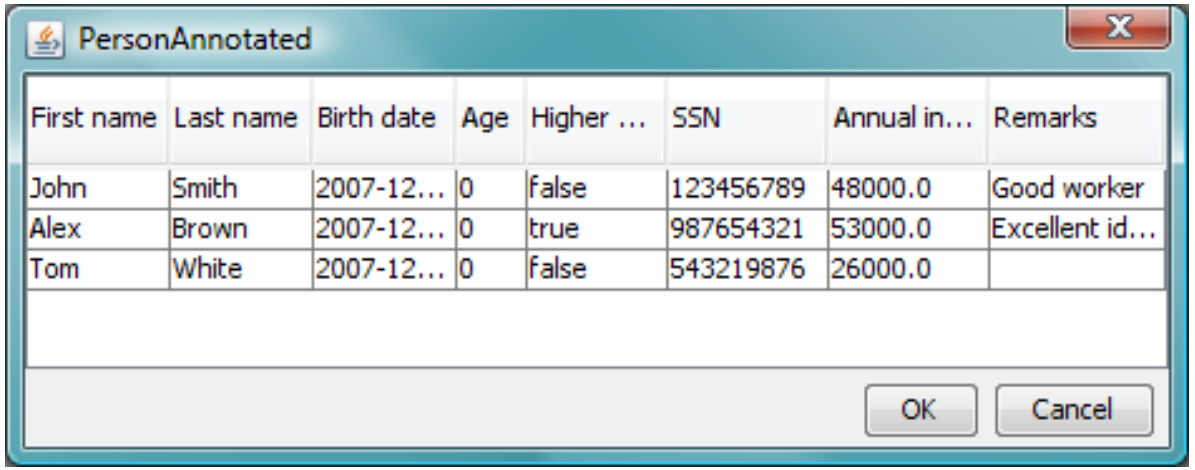
Połączenie z `ObjectPlusX`

- Dzięki połączeniu senseGUI razem z wcześniej omawianą biblioteką `ObjectPlusX` otrzymujemy również możliwość zarządzania powiązaniami, ekstensją, itp.



Dodatkowe możliwości

- Wyszukiwanie obiektów
  - Kryteria wprowadzamy za pomocą wygenerowanej formatki
  - Automatyczna informacja o postępach
- Widok tabelaryczny
  - Dowolna kolekcja,
  - Możliwość wyboru obiektu z tabeli.



Fluent API zamiast adnotacji

- Zamiast adnotacji, można stworzyć własny język dziedzinowy (*DSL – Domain Specific Language*).
- Jedną z możliwości jest wykorzystanie tzw. *Fluent API*.
- Popularne rozwiązania, np.
  - jQuery,
  - LINQ,
  - mapowanie modelu w Entity Framework.

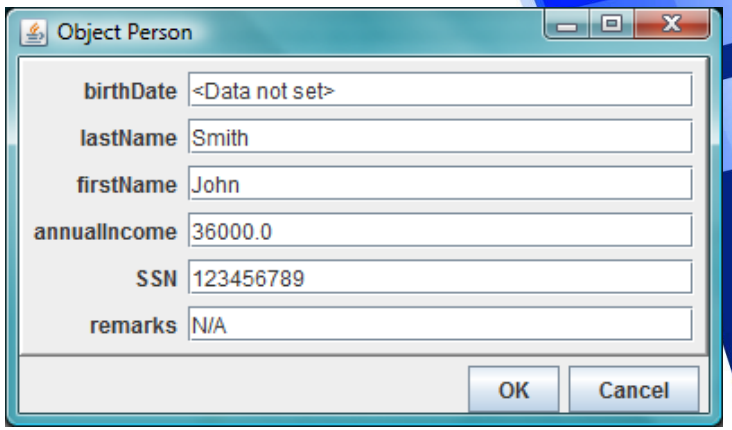
```
using (var ctx = new BloggingContext()) {
  var blogs = ctx.Blogs.Include(blog => blog.Posts).ToList();
}
```

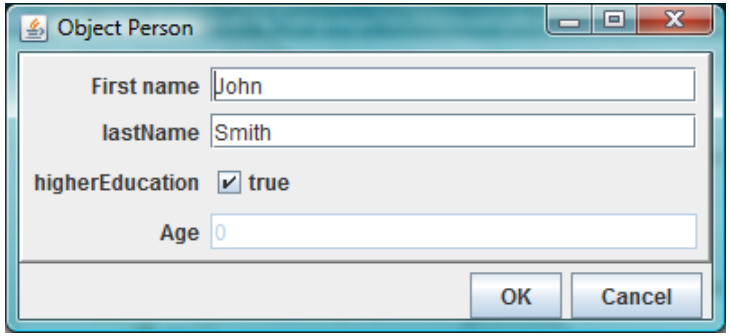
GCL DSL

- Nowa wersja biblioteki:
  - Fluent API zamiast adnotacji,
  - Walidatory,
  - *I18n*,
  - *AdHoc* GUI,
  - https://code.google.com/archive/p/gcl-dsl/
  - *Trzaska M.: Data Migration and Validation Using the Smart Persistence Layer 2.0. The 16th IASTED International Conference on Software Engineering and Applications (SEA). Las Vegas, USA. Acta Press. ISBN: 978-0-88986-951-6. Pobierz*
  - *Trzaska M. GCL - An Easy Way for Creating Graphical User Interfaces. Journal of Systemics, Cybernetics and Informatics. ISSN: 1690-4524. pp. 81-88. Pobierz*

GCL DSL (2)

```
JFrame frame1
= create.frame.usingOnly(person);
```

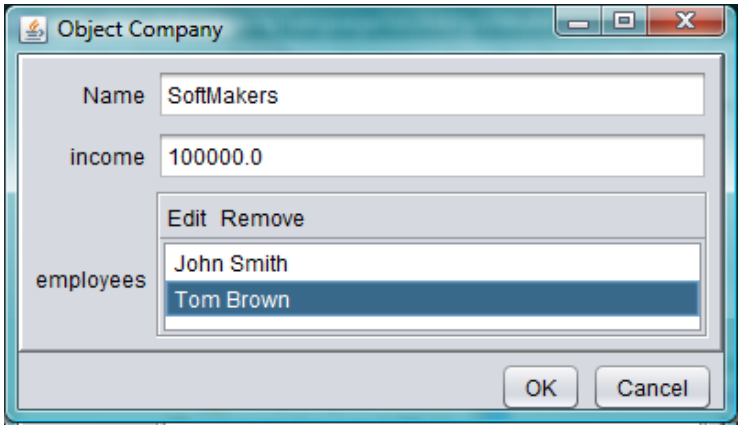




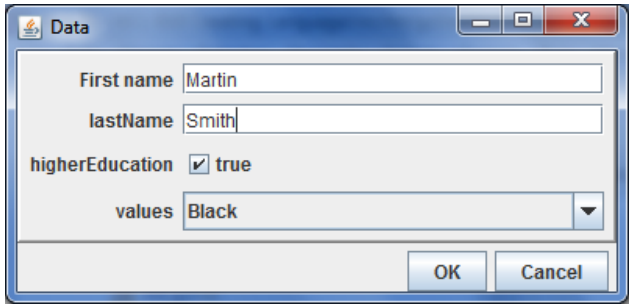

```
JFrame frame =  create.
                frame.
                using(person).
                containing(
                    attribute("firstName").as("First name"),
                    attribute("lastName").validate(new ValidatorNotEmpty()),
                    attribute("higherEducation"),
                    method("getAge").as("Age"));
```

GCL DSL (3)

```
JFrame frame =  create.
                frame.
                using(company).
                containing(
                        attribute("name").as("Name"),
                         attribute("income"),
                         attribute("employees"));
```




```
AdHocActionPerformed processAccept = new AdHocActionPerformed() {
        @Override
        public void Accept(Map<String, String> enteredData) {
                // Do something with the fields...
        }
};
frame =  create.
        frame("Data", processAccept, "OK").
        containing(
                attribute("firstName").as("First name").value("Martin"),
                attribute("lastName").validate(new ValidatorNotEmpty()),
                attribute("higherEducation").type(boolean.class),
                attribute("values").type(Colors.class));
```



Podsumowanie

- Wydaje się, że deklaratywne podejście do tworzenia GUI jest bardzo użyteczne.
- Dzięki niemu programista jest w stanie zaoszczędzić sporo pracy – szczególnie w przypadku typowych formularzy.
- Każde tego typu rozwiązanie jest kompromisem pomiędzy zaangażowaniem programisty, a osiągniętymi efektami.
- Zademonstrowane przykłady pokazują, że da się je wykorzystać w praktyce.