Week 7: Case study: Designing, building, & testing a Person class

- Background for assignment #5
- Needed in many applications
- Is it possible? practical?

Question

- Is it possible to define a Person class that will support all those application systems?
  - If so, is it desirable?
  - worth doing?

- Some experts say: No.
  - The needs of each application area are unique.
  - You'll just distort the application if you try.

- Others say: Of course.
  - Otherwise what good is OOP?
  
  What do you think?

Can we exploit inheritance?

- Many applications deal with records representing people;
- For example:
  - Personnel systems (Employee)
  - Medical systems (Patient)
  - Academic systems (Student)
  - Library systems (Author)
  - Billing systems (Customer)

An Employee is a Person, etc.
Obstacles to inheritance

- Suppose the same Person is both a Student and an Employee
  - Then we'd have two objects
  - Would they be equal? Should they be?
- How would we specify accessibility?
  - Number of dependents should be public for an Employee,
  - But none of anyone's business for a Customer
- The "subclasses" aren't really special kinds of Person.
  - They're roles that a Person may play.

Relationship between Role and Person

- Natural language is misleading. It sounds reasonable to say that a Student is a Person.
- But actually:
  - A Person plays a Role (maybe more than one)
  - A Role is assigned to a Person.
- In OOP those are best represented by a has-a (membership) relationship, even though that departs from common usage.

Breakthrough!

- Now we can focus on designing a Person class without having to be concerned about roles.
- A Person object should represent properties that are common to all Persons and are relatively stable.

Basic Person properties

<table>
<thead>
<tr>
<th>Permanent properties</th>
<th>Change is extremely rare</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateOfBirth</td>
<td></td>
</tr>
<tr>
<td>name</td>
<td></td>
</tr>
<tr>
<td>sex</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stable properties</th>
<th>Change is infrequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>maritalStatus</td>
<td></td>
</tr>
<tr>
<td>citizenship</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transient properties</th>
<th>Change may be frequent</th>
</tr>
</thead>
<tbody>
<tr>
<td>mailingAddress</td>
<td></td>
</tr>
<tr>
<td>telephoneNumber</td>
<td></td>
</tr>
</tbody>
</table>

What are those properties? What else?
Let's focus on the permanent properties

- The other properties can be delegated to subclasses
- role classes
- databases

- A general Person class can be useful in most applications.

The basic Person class

- Member data items
  - name
  - dateOfBirth
  - dateOfDeath
  - sex

- Questions:
  - What's missing?
  - What data type is each member?

The identity problem

- Suppose we have
  - `personA`
    - name = "George Jefferson"
    - sex = male
    - dateOfBirth = June 14, 1964
    - citizenship = USA
  - `personB`
    - name = "George Jefferson"
    - sex = male
    - dateOfBirth = June 14, 1964
    - citizenship = USA

- Q: What should `personA == (personB)` return?

The identifier problem

- The Boolean equals predicate (Java) or the `==` operator (C++, C#) must return:
  - true, or
  - false

- It may not return
  - maybe, or
  - probably

- Therefore each Person object must have a unique identifier
  - Who assigns them?
  - What's their scope?
Many organizations assign their own, often specific to an application:
- customer number
- student number

But we still can't tell if a student and a customer are the same person.

In the United States the social security number (SSN) is the only universal ID
- It's not supposed to be used for other purposes.
- But everyone does, anyway.
- And you have to apply for one

A Person object contains an ID field

The ID can be anything, as long as it's unique within the scope of the applications that encounter it.

Some versions may allow two ID fields
- An interim one, if the permanent one is unknown at the time the object is first created.
- A permanent one, once we know it.

Let's focus on RealPerson

Every LivingPerson eventually becomes a DeceasedPerson

So let's not bother with a cumbersome subclass arrangement.
- Just include a dateOfDeath field, which is null for a LivingPerson.
A Person class
(rough version)

```java
public class Person {
    // What are the missing type codes?
    long ID;
    String name;
    Date dateOfBirth;
    Date dateOfDeath;
    Boolean sex;

    // Constructors
    // Accessors
    // Relational operators
}
```

A naive version

```java
public class Person {
    long ID;
    String name;
    Date dateOfBirth;
    Date dateOfDeath;
    Boolean sex;

    // Constructors
    // Accessors
    // Relational operators
}
```

What's wrong with that?

We're trying to use the object-oriented paradigm

```java
public class Person {
    long ID;
    String name;
    Date dateOfBirth;
    Date dateOfDeath;
    Boolean sex;

    // Constructors
    // Accessors
    // Relational operators
}
```

Which of those members should be objects?

Consider a Person's name

- What components does it have?
- What format is it in?
  - English (first middle last)
  - Directory (last, first middle)
  - Other
- How long may it be?
  - Will it fit on a mailing label?
- Can a raw `String` satisfy those criteria?
- Can a Person's name appear elsewhere (not in a Person object)? Examples?
Conclusion
- We need a **PersonName** class in order to:
  - standardize and enforce internal representation
  - facilitate data entry and retrieval
- It must be a top level class, independent of **Person**.
- See
  [http://www.idinews.com/personName.html](http://www.idinews.com/personName.html) for a start.

What about Sex? or Gender
- Three possible values:
  - male
  - female
  - unknown
  - **That rules out** **Boolean**!
  - **What about an integer?**
    - a single character (M, F, U)?
- Must it be standard
  - for an organization?
  - within a single application system?
  - throughout the world?

Another class?
- It looks as if we'll need another class for **Sex** (or Gender).
- Since it has only three possible values and they're usually immutable, we should consider the **enum** facility.

Are we done yet?
- What about the **identifier** field?
  - We tentatively chose **long**, but some identifiers contain non-numeric characters.
  - We can use a **String**, but then there's no central format control or enforcement
- So we need a **PersonID** class.
- **Q:** Aren't we proliferating too many classes?
- **A:** No, that's what OOP is about.
An improved version

```java
public class Person {
    PersonID ID;
    PersonName name;
    Date dateOfBirth;
    Date dateOfDeath;
    Sex sex;

    // Constructors
    // Accessors
    // Relational operators
}
```

More bad news for Java programmers!

- **Date** and **Calendar** are among the few numeric classes in the standard Java library. We'd like to use them if we can.

- But those are among the worst designed components of the standard Java library! They're extremely awkward, inflexible, and error prone.

  In what ways?

  - In any Java application where date manipulation and calculation are important, therefore, we prefer to use our own **Date** class.

  - Fortunately C#'s is better, but you may still prefer to develop your own.

The bottom line (work plan)

- We need to develop (design, code, and test) classes, probably in bottom-up sequence.
  - PersonID
  - PersonName
  - Date
  - Sex
  - and finally Person.

- Isn't that an awful lot of work?
  - Yes, but all of those classes are potentially reusable and candidates for standardization at least within an organization.
  - That justifies the investment.

Unfortunately

- Few programming organizations bother to develop a solid standard way of representing a **Person**.
  - They may not bother with a **Person** class at all.
  - or they may rely on a badly flawed early version.

- Does that have a significant impact on software quality?

- If so, what should Q.A. do about it in an organization?