AARHUS UNIVERSITET

Software Engineering and Architecture

Bloch Builder

AARHUS UNIVERSITET

Problem

- We need to define *PersonIdentity* objects
  - Representing
    - Names, SSN (CPR), phone numbers, gender, date of birth, address, title, …
  - Moreover, need to define *partial knowledge*
    - I.e. ‘Per Christensen’ but we do not know date of birth, phone number, etc.
  - And potential with *complex constraints*
    - Ex. Assign title ‘Mr.’ to male, ‘Miss’ to female, in case no title is assigned
  - And they are *immutable objects*
- *Objects with complex state to be set*

AARHUS UNIVERSITET

Problem

- The solution is well known: **Constructor parameters…**

```
public class PersonIdentityVanilla {

  private String ssn;
  private String prefix;
  private String[] givenNames;
  private String familyName;
  private Gender gender;
  private Date birthTime;

  private String[] phoneNumbers;


  public PersonIdentityVanilla(String familyName, String prefix,
      String[] givenNames, Gender gender, String personID, String[] phoneNumbers,
      int birthYear, int birthMonth, int birthDayInMonth) {
    this.familyName = familyName;
    this.prefix = prefix;
    this.gender = gender;
    this.ssn = personID;
    this.givenNames = givenNames;
    this.phoneNumbers = phoneNumbers;
    if (birthYear != 0) {
      Calendar utcCalendar = Calendar.getInstance(TimeZone.getTimeZone("UTC"));
      utcCalendar.set(birthYear, birthMonth, birthDayInMonth, 0, 0, 0);
      birthTime = utcCalendar.getTime();
    }
  }
```

AARHUS UNIVERSITET

Usage…

- Defining John and Ann…
  - Exercise: Liabilities ???

```
public static void main(String[] args) {
  System.out.println( "======= Demonstration of Vanilla Constructor =======");

  // Define the identify of the person 'John William Hansen'
  PersonIdentityVanilla john =
    new PersonIdentityVanilla("Hansen", "Mr.",
                              new String[] { "John", "William" },
                              Gender.Male,
                              "171100-1357",
                              new String[] {"+45 9812 3456", "+45 9812 5687"},
                              1998, Calendar.NOVEMBER, 17
    );
  System.out.println(john.toString());

  // Define the identify of the person 'Ann Nielsen'
  PersonIdentityVanilla ann =
      new PersonIdentityVanilla("Nielsen", "Dr.",
          new String[] { "Ann" },
          Gender.Female,
          null,
          null,
          0, 0, 0);

  System.out.println("=== Ann: ===");
  System.out.println(ann.toString());
}
```

```
======= Demonstration of Vanilla Constructor =======
=== John: ===
PersonIdentity:
 Title         : Mr.
 Family name   : Hansen
 Given name(s) : John / William /
 Gender        : Male
 SSN/ID        : 171100-1357
 Birth         : Tue Nov 17 01:00:00 CET 1998
 Phone         : [+45 9812 3456,+45 9812 5687,]
---
=== Ann: ===
PersonIdentity:
 Title         : Dr.
 Family name   : Nielsen
 Given name(s) : Ann /
 Gender        : Female
 SSN/ID        : null
 Birth         : null
---
```

Exercise

- An alternative would be
  - PersonIdentity pi = new PersonIdentity();
  - pi.setFamilyName(”Hansen”);
  - pi.addGivenName(”Kaj”);
  - pi.setGender(Gender.Male);
  - …

- Why is this not an acceptable solution?

AARHUS UNIVERSITET

Bloch's Builder / Effective Java

- Alternative
  - Construct a *builder* object, using its *setter* methods, and finally ask it to *build* the PersonIdentity

```
public static void main(String[] args) {
  System.out.println( "======= Demonstration of Bloch's Builder =======");

  // Define the identify of the person 'John Hansen'
  PersonIdentity john =
    new PersonIdentity.PersonBuilder("Hansen").
    setPrefix("Mr.").
    addGivenName("John").
    addGivenName("William").
    setGender(Gender.Male).
    setPersonID("171100-1357").
    addPhoneNumber("+45 9812 3456").
    addPhoneNumber("+45 9812 5687").
    setBirthTime(1998, Calendar.NOVEMBER, 17).
    build();

  System.out.println("=== John: ===");
  System.out.println(john.toString());
```

```
// Define the identify of the person 'Ann Nielsen'
PersonIdentity ann =
  new PersonIdentity.PersonBuilder().
  setGender(Gender.Female).
  setPrefix("Dr.").
  addGivenName("Ann").
  setFamilyName("Nielsen").
  build();

System.out.println("=== Ann: ===");
System.out.println(ann.toString());
```

AARHUS UNIVERSITET

Bloch's Builder / Effective Java

- The internal Builder

```
// Define the identify of the person 'Ann Nielsen'
PersonIdentity ann =
  new PersonIdentity.PersonBuilder().
  setGender(Gender.Female).
```

```
public static class PersonBuilder {
  // Required parameters
  private String familyName;
  private String[] givenNames;

  // Optional parameters
  private String patientId = null;
  private Gender gender = Gender.Undifferentiated;
  private Date birthTime = null;
  private String[] phoneNumberArray = null;
  private String prefix = null;

  // temporaries
  private int yearAllDigits = 0;
  private int dayInMonth = 0;
  private int month = 0;
  private List<String> givenNamesTemporary;
  private List<String> phoneNumberList;

  public PersonBuilder() {
    givenNamesTemporary = new ArrayList<String>();
    phoneNumberList = new ArrayList<String>();
  }
```

```
public PersonBuilder addGivenName(String name) {
  givenNamesTemporary.add(name);
  return this;
}
public PersonBuilder setPersonID(String patientId) {
  this.patientId = patientId;
  return this;
}
public PersonBuilder setGender(Gender gender) {
  this.gender = gender;
  return this;
}
```

AARHUS UNIVERSITET

Bloch's Builder / Effective Java

- The **build()** method

```
public PersonIdentity build() {
  givenNames = new String[givenNamesTemporary.size()];
  givenNamesTemporary.toArray(givenNames);

  if (phoneNumberList.size() > 0) {
    phoneNumberArray = new String[ phoneNumberList.size()];
    phoneNumberList.toArray(phoneNumberArray);
  }

  if (yearAllDigits != 0) {
    Calendar utcCalendar =
      Calendar.getInstance(TimeZone.getTimeZone("UTC"));
    utcCalendar.set(yearAllDigits, month, dayInMonth, 0, 0, 0);
    birthTime = utcCalendar.getTime();
  }
  PersonIdentity pi = new PersonIdentity(this);
  return pi;
}
```

- Depends on *private* constructor

```
private PersonIdentity(PersonBuilder builder) {
  familyName = builder.familyName;
  givenNames = builder.givenNames;
  ssn = builder.patientId;
  gender = builder.gender;
  birthTime = builder.birthTime;
  phoneNumbers = builder.phoneNumberArray;
  prefix = builder.prefix;
}
```

AARHUS UNIVERSITET

The Two-Phase Process

- The algorithm
  - You create the *internal builder object*
  - You set all state, using its setters, in temporary state variables, in the internal builder
  - You invoke its *build()*
    - Which can
      - Verify constraints between state variables
      - Set/alter additional state as defined by requirements
    - Compute the final state space, and…
  - … that invokes the final object's constructor
    - That copies from the builder's state into its own state variables

3-1-2 ???

- Hm…

- (2) Favor object composition
  - Two objects collaborate to produce a single immutable object

AARHUS UNIVERSITET

Another Example

- EtaStone ‘Effect’s
  - When a card is played, it executes an ‘effect’
- Old ‘direct’ code for EtaStone’s TomatoSalad

Tomato Salad	(2, 2, 2)	Add +1 attack to random own minion	“M: (+1,0)”

```
new CardSpec(GameConstants.TOMATO_SALAD_CARD, mana: 2, attack: 3, health: 2,
        internalModifiableGame -> {
          Player who = internalModifiableGame.getPlayerInTurn();
          int max = internalModifiableGame.getFieldSize(who);
          if (max == 0) return; // No fielded card, abort
          int n = randomNumberStrategy.computeRandomNumber(max);
          internalModifiableGame.deltaAttackMinion(who, n, delta: +1);
        }, effectDescription: "M: (+1,0)", Attributable.NONE),
```

Effect

- New code, using a *Fluent API*

```
new CardSpec(GameConstants.TOMATO_SALAD_CARD, mana: 2, attack: 3, health: 2,
        internalModifiableGame -> {
          new Effect(internalModifiableGame)
                  .forMe()
                  .forRandomMinion(randomNumberStrategy)
                  .deltaAttack(+1);
        }, effectDescription: "M: (+1,0)", Attributable.NONE),
```

AARHUS UNIVERSITET

Other Examples

- From my SigmaStone

```
// 2 dmg to all opp minion, +2 health to all own
new CardSpec(PlayKata23Constants.HOLY_NOVA, mana: 3, attack: 0, health: 0,
        mutGame -> {
          new Effect(mutGame).forMe().forAll().deltaHealthOrKill(+2);
          new Effect(mutGame).forOpponent().forAll().deltaHealthOrKill(-2);
        },
         effectDescription: "HolyMoly", Attributable.NONE),
```

- The point
  - Readable and very short

AARHUS UNIVERSITET

Analysis

- Benefits
  - Improved analyzability of complex object construction *(fluent API)*
  - Improved reliability (lower probability of wrong coding)
  - Improved support for partial object construction
  - Improved support for constraint checking
- Liabilities
  - Complex coding of the internal builder
    - Intermediate state objects
    - Lots of named setters
    - Private constructor in the outer/final object
- Conclusion: Careful evaluate benefit/liabilities

AARHUS UNIVERSITET

Why do you need to know?

- This builder is increasingly seen in open source libraries

```
Unirest.post("http://httpbin.org/post")
  .queryString("name", "Mark")
  .field("last", "Polo")
  .asJson()
```

- Example:
  - Unirest
    - asJson() builds the http request and executes it
  - TestContainers
    - Build docker containers
  - Logging frameworks, Mock frameworks, …

```
// Given 'cmd' container on the crunch network, with
GenericContainer cmd =
        new GenericContainer<>( dockerImageName: "henrikbaerbak/private:cave")
                .withNetwork(network)
                // the 'doc.cpf' as CPF (tells that the daemon is on the crunch network
                .withCopyFileToContainer(MountableFile.forClasspathResource("cpf/doc.cp
                        containerPath: "/root/cave/client/src/main/resources/cpf/doc.cpf")
                // and a list of commands for cmd to execute
                .withCopyFileToContainer(MountableFile.forClasspathResource("crunch/cmd
                        containerPath: "/root/cave/cmdExec1.txt")
                // and mounting the .gradle folder on the host (HARDCODING)
                .withFileSystemBind( hostPath: "/home/csdev/.gradle-crunch",  containerPath:
                // and these provided as input to Cmd
                .withCommand("./gradlew", ":client:cmd", "-Pcpf=doc.cpf", "-Pcmdlistfil
                // And await until cmd is ready for input
                .waitingFor(
                        Wait.forLogMessage( regex: "Welcome to SkyCave",  times: 1)
                )
                // or at least 2 minutes
                .withStartupTimeout(Duration.ofMinutes(2))
```

AARHUS UNIVERSITET

Relation to GoF Builder

- Bloch states that this pattern is equal to GoF's Builder pattern
  - *I do not really agree…*
    - The Intent is different. GoF Builder intent: "*… so the same construction process can create different representations*"

Intent	Separate the construction of a complex object from its representation so that the same construction process can create different representations.

- Bloch's builder intent is to create *single representation* but solves *faced with many constructor parameters*

Reference

- Great book ☺