# Eve Programming

# Guidelines for Java Programming

Anton Liebetrau, iSolutions (VICS 3)
Version 1.02, December 2001

Translation/English version:
George Raymond / 16 May 2001

**Inhaltsverzeichnis**

# 1. Introduction

In the course of our practical work on the projects "OS Bank", "iqs 4 insurancelab" and "wincolink planet", we developed coding and naming conventions that – for example when training new co-workers – have proved to be very useful. To make our experience available to everyone, we have compiled it in the guidelines you are now reading. In this context, the proposed conventions and described "tips and tricks" support the following goals:

- Facilitate training of new co-workers
- Increase the readability of program code
- Increase the clarity of classes and methods
- Simplify application maintenance
- Enable extension of applications
- Increase the stability of applications

## 1.1 Important literature

Of course, a large number of software developers throughout the world have already reached the same conclusions we have. We would therefore like to mention here at the start two very interesting (and for us important) publications:

- [Sun1999] – Sun team: *Code Conventions for the Java Programming Language*, Sun Microsystems, 1999 (we have this as a PDF document; for the original see in the Internet under http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html). – **Contents:** Coding guidelines and naming conventions that we approve, and largely adhere to.
- [RogueWave2000] – Rogue Wave team: *The Elements of Java Style*, Cambridge University Press, 2000. – **Contents:** More than 100 recommendations, tips and tricks for programming in Java (topics: coding, naming, documentation, programming, and creation of packages).

**Important:** We strongly recommend that you read these two documents in parallel to the present one. They contain very interesting material for any software developer, and we have adopted a number of guidelines without modification (particularly the coding guidelines in [Sun1999]).

## 1.2 Truisms

Unfortunately, if you are a clever, experienced and therefore sought-after software developer, you will often encounter tips in these guidelines that you have already long known (such a tip is known as a truism[1]). I would therefore like excuse myself in advance for any tired chuckling, impatient rustling, or hoarse throat-clearing that I may generate – sorry. But this document is for all newcomers, upward climbers, and experts who want to know how we in the Team I-Solutions (VICS 3) develop high-quality software and what guidelines we follow.[2]

[1] A truism is a well-known fact. (This relates to the Latin expression, *nodum in scirpo quaerere*: to search for a knob on the – smooth – stem of a rush plant, meaning to look for difficulties where there are none.)
[2] Kent Beck comments on this topic as follows (see [Beck2000], page 61): "If you are going to have all these programmers [...] swapping partners a couple of times a day, and refactoring each other's code constantly, you simply cannot afford to have different sets of coding practices. With a little practice, it should become impossible to say who on the team wrote what code."

My two favorite mottoes are: "short and sweet" and "better redundant than unclear". In writing these guidelines, I hope I have succeeded in giving both of these (contradictory) mottoes enough attention.

### 1.3 Project "Eve"

Eve has now been living in Paradise[3] for quite some time; according to Jaermann/Schaad, she is currently working at the checkout of the Cosmos department store (and is daily to be seen in the Zurich newspaper *Tages-Anzeiger*).


**Figure 1: "In love!"... "Engaged!" ... "4 francs 50":** Eve on Valentines Day 2001 at the checkout of the Cosmos department store (Jaermann/Schaad in the *Tages-Anzeiger* of February 14, 2001).

Project "Eve" is to produce a whole series of short, information-packed documents offering a paradise for the IT specialist. These documents should provide solid support for software developers, project managers, and testers in their daily work. The following topics will be among those covered:

- Methodology for projects
- Guidelines for Java programming
- Testing Java programs
- The Eve application framework

### 1.4 I would like to thank …

… all those members of the teams *VICS 11, VINS 17* and *VICS 3* who, with their comments and questions, mercilessly brought to light the (naturally long-since eliminated) weaknesses of this document and who pressured me to provide additional or more-precise information. Thanks, VINS 171–179, VICS 110–119 and VICS 300–399, maybe you'll discover still other weak points …

[3] Paradise is generally understood as a place or area that offers ideal conditions for a certain group of people or living things.

# 2. Coding guidelines

By coding guidelines, we mean formatting rules for Java source code. In the coding guidelines, we have sought wherever possible to stick to the recommendations of *Sun* [Sun1999], although we also approve the recommendations of *Rogue Wave* [RogueWave2000] (even though they differ slightly from Sun's).

## 2.1 Automatic formatting

It is well known that in the development tool VisualAge, you can activate automatic formatting of source code in *Options*, in the category *Coding, Formatter*. You can call up this dialog in any window by using the menu item *Window, Options...* It looks at follows:

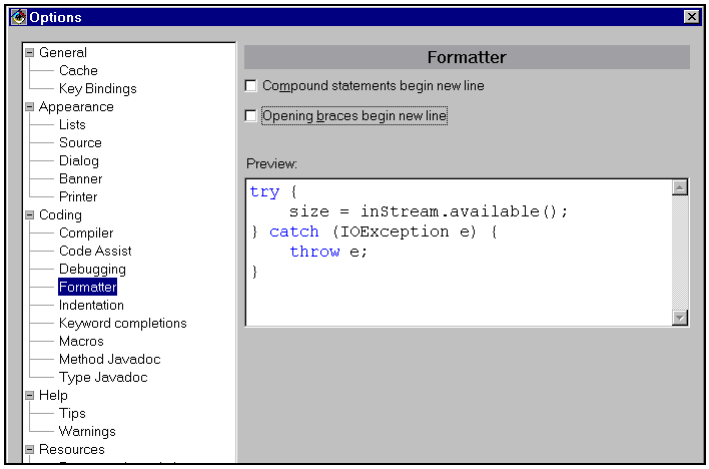


**Figure 2:** Dialog *Options*, Category *Coding, Formatter*. Here, you can use the two checkboxes *Compound statements ...* and *Opening braces ...* to set the desired formatting of the source code.

Within the editor window, VisualAge automatically formats the source code. If you now use the *Delete* or *Backspace* key, however, you can easily make a mess of the formatting. Using the command *Format Code* of the Editor context menu, you can format the source code anew at any time:

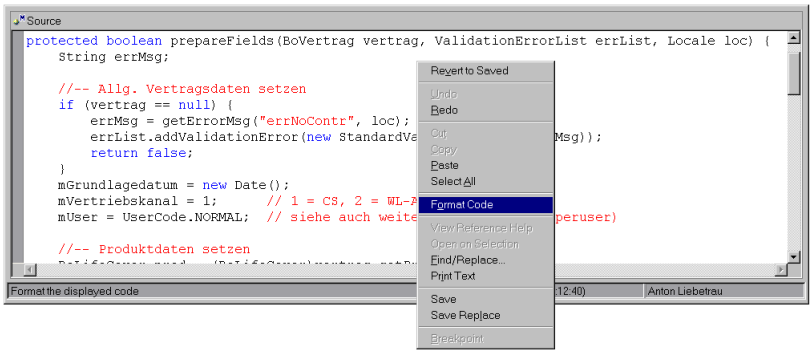


**Figure 3:** Using the menu item *Format Code* of the Editor context menu, you can impose the pre-set formatting type at any time.

The following three figures show how you can set the two checkboxes in the category *Coding, Formatter* of the dialog *Options*, to obtain particular formats:

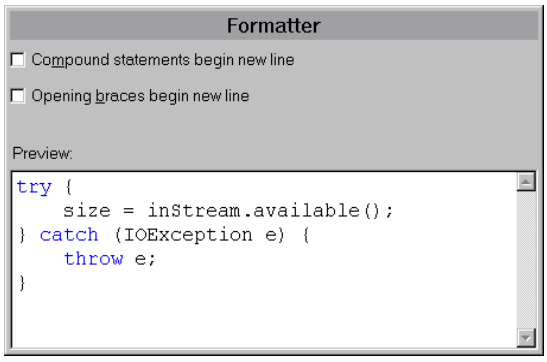


**Figure 4:** The formatting we prefer, according to the recommendations of Sun.

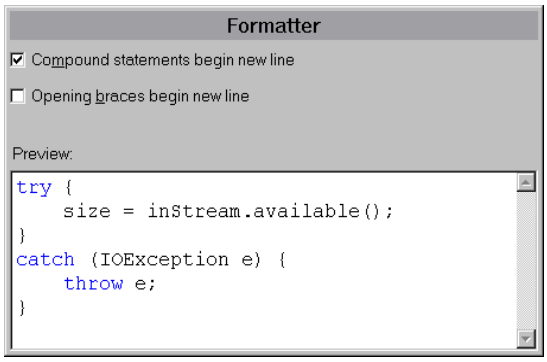


**Figure 5:** Formatting according to the recommendations of the company Rogue Wave.

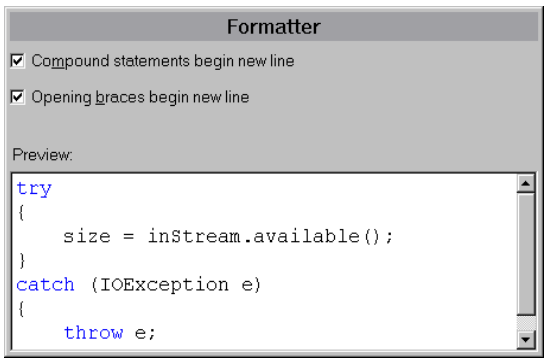


**Figure 6:** Formatting for inveterate C++ programmers.

As these three figures show, in the realm of code formatting, there are several versions of the truth – each formatting style has its pros and cons. In general – as already mentioned several times – we prefer the formatting of Sun, but also respect the Rogue Wave and C++ formatting styles.

### 2.2 Consistency

Within the methods of a particular class, all programmers must use a consistent code formatting style (to promote legibility). Thus, if a Rogue Wave enthusiast works on the class of a Sun fan, the Rogue Waver must use Sun formatting – and vice versa in the opposite case.

# 3. Naming conventions

By naming conventions, we mean rules for writing the names of packages, classes, methods and attributes.

## 3.1 Packages

You should generally write package names in all lower-case letters.[4] Just as a folder can contain several sub-folders, a package can contain several sub-packages. Package names should be short and meaningful, and have the following structure:

```
com.winterthur."project_name"."project_package"["project_sub-package"][...]
```

All our package names must start with the (sub-)packages *com* and *winterthur*.

Possible package names for the project "Jackpot":

```
com.winterthur.jackpot
com.winterthur.jackpot.util
com.winterthur.jackpot.servlets
com.winterthur.jackpot.servlets.states
```

Possible package names for the project "Eve":

```
com.winterthur.eve.util
com.winterthur.eve.printing
com.winterthur.eve.calculation
com.winterthur.eve.calculation.corba
com.winterthur.eve.calculation.rmi
```

In contrast to Java, the VisualAge development environment offers an additional organizational level, the "project", in which you can place a group of related packages. As the following figure shows, a main project contains a number of sub-projects:

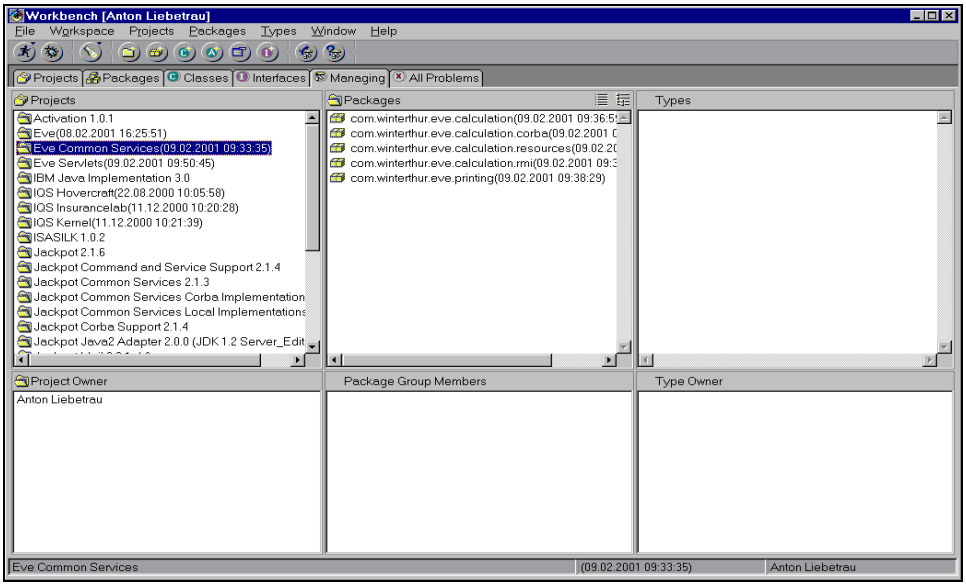


**Figure 7:** Main projects typically consist of several sub-projects. For example, the main project "Eve" contains the three sub-projects "Eve" (same name as the main project), "Eve Common Services" and "Eve Servlets".

[4] Certain tools create identical class and package names (for example, the stub and skeleton generator of CORBA). Given that every class must begin with a capital letter, the automatically-generated package names also start with a capital letter (this is the only allowable exception).

Sub-projects do not necessarily need to be visible in the main project structure. (To promote manageability, you should make sub-projects visible only if they really need to be.)

For example:

| Project | Packages within project |
|---|---|
| Eve | com.winterthur.eve.util<br>com.winterthur.eve.codetables<br>com.winterthur.eve.codetables.filter |
| Eve Servlets | com.winterthur.eve.servlets.states<br>com.winterthur.eve.servlets.util |
| Eve Common Services | com.winterthur.eve.printing<br>com.winterthur.eve.calculation<br>com.winterthur.eve.calculation.corba<br>com.winterthur.eve.calculation.rmi |

**See also:** Sections 15–17 in [RogueWave2000].

### 3.2 Classes and interfaces

Class and interface names always start with a capital letter; the rest of the letters are lower-case. Capital letters can be used within a class or interface name to enhance its legibility. (The underscore character ("_") should only be used when absolutely necessary or only when a framework requires it. The following classes *ResPapyrus* provide an example.) Some examples:

```
BoContract
BoNaturalPerson
LifeFundQuotation
LifeFundController
LifeFundQuotationEntry
ResPapyrus       // Basic class for text resources
ResPapyrus_de    // German text resources
ResPapyrus_fr    // French text resources
```

**See also:** Sections 18-21 in [RogueWave2000].

### 3.3 Methods

Method names always start with a lower-case letter (except of course for constructor names, which always have the same name of the class and therefore must start with a capital letter). Within a method name, you can use capital letters to enhance legibility. Avoid the underscore character ("_") whenever possible. It is best if the method name indicates the data type of the return value (this is, however, not always possible and doesn't always make sense). Some examples:

```
needJumpNameInstance(...) // Return value type: a JumpName instance
needProductName(...)      // Return value type: probably a string
prepareFields(...)        // Return value type: void, boolean ... (?)
trimArray(...)            // Return value type: an array or void
perform(...)              // Return value type: void, boolean ... (?)
asString(...)             // Return value type: string
readDate(...)             // Return value type: date
isParent(...)             // Return value type: boolean
hasParent(...)            // Return value type: boolean
canClose(...)             // Return value type: boolean
```

**Names of boolean methods:** An *is*, *has* or *can* method must return a boolean value, but a method with a boolean return value need not necessarily start with *is*, *has* or *can*. – To make this clearer: Suppose the method *perform()* calculates data for an insurance product and returns a boolean value showing that it was able to successfully carry out the calculation. The main task of this method is thus the calculation, and not the validation of the product. The prefix *is*, *has* or *can* is therefore missing from this method's name. (The boolean return value is a sort of by-product and not the main purpose of this method.)

**See also:** Within this document, section 5.2, JavaBean convention, and sections 22–24 in [RogueWave2000].

### 3.4 Attributes

In practice it has proved useful to specially mark class attributes (also known as fields, or member variables) to enhance legibility. We can distinguish between three different attribute-types:

- constants (static, final attributes)
- class variables (static attributes)
- instance variables (all other attributes)

#### 3.4.1 Constants

Constants are always written in ALL CAPITAL letters. The underscore ("_") serves to break up long names. Some examples:

```
//-- Constants (= static final)
public static final int ERR_OK = 0;
public static final int ERR_LOADPRODUCT = 1;
public static final String STATENAME = "LifeFundFundList";
public static final String QUOTATION_VNVP_QUERY = "VnEqualsVpQuery";
```

#### 3.4.2 Class variables

Class variables receive the prefix "mst" and are generally written in lower-case letters ("m" stands for *member*, "st" for *static*). You can use capital letters to break up long names. Some examples:

```
//-- Class variables (= static)
private static Boolean mstBigEndian = null;
private static int[] mstErrorList = new int[5];
private static InsurancelabProperties mstMySelf = null;
```

#### 3.4.3 Instance variables

Instance variables receive the prefix "m" and are generally written in lower-case letters ("m" stands for *member*). You can use capital letters to break up long names. Some examples:

```
//-- Instance variables
private Object mInputData;
private String mProjectPath;
private long mVmId;
private DataOutputStream mDos;
private DataInputStream mDis;
```

**See also:** Within this document, section 5.1, Generation of access methods, and sections 25-31 in [RogueWave2000].

# 4. Comments

Comments are indispensable for understanding programming code, but are often neglected by programmers. There are many reasons for this: Lack of time due to deadline pressure, general lack of interest, emerging wisdom teeth, itchy scalp, and tingling toenails. – To write clear and useful comments, you need to follow just a few rules:

## 4.1 Only describe the essential

In comments, address the essential issues only. References to other classes and methods are often better than long-winded descriptions.

```
/**
 * Good: States the essential (mentions the method "perform", which is needed for
 * the calculation).
 *
 * This class is always needed when data for a product is to be calculated
 * (with the help of the method "perform"). The user of this class
 * does not know whether the calculation occurs by way of RMI, CORBA or a
 * direct connection
 * (= transparent type of call; see also the method "getCalculator").
 *
 * Creation date: (28 Feb 2001)
 * @author: John Johnson
 */

/**
 * Bad: Describes processes (that could change), mentions too many
 * details (that don't belong here), doesn't mention the most important thing
 * (the method "perform", which is triggered with the calculation).
 *
 * This class is needed for the calculation of data for a product. In this
 * context, first of all "getCalculator" is automatically called. There, with the
 * help of the Properties file "calcitnow.properties", it is determined whether
 * the premium calculator is to be called by way of RMI, CORBA or a direct
 * connection (in case of an error, the exception "PropFileNotFoundException" or
 * "WrongCalculatorStyleException" is triggered). Then the calculation can at
 * last occur; here again, the exceptions "WrongDataException" or "CalcException"
 * can be triggered.
 *
 * Creation date: (28 Feb 2001)
 * @author: John Johnson
 */
```

## 4.2 Comments in classes and methods – "What does it do?"

General comments in a method or class should make clear *what it does* – and not *how it does it* (the "how" can change frequently, the "what" much less often):

```
/**
 * Good: Describes "what".
 *
 * This method starts with the three separate strings "year", "month" and
 * "day", generates a date, and returns it. If, on the basis of the parameters,
 * the method cannot generate a valid date, it returns "null".
 *
 * Creation date: (28 Feb 2001)
 * @author: John Johnson
 */

/**
 * Bad: In addition to "what", also describes "how".
 *
 * This method starts with the three separate strings "year", "month" and
 * "day", creates a date, and returns it. To generate the date, the method
 * first transforms the input strings into int values. The method then
 * uses the class "GregorianCalendar" to convert these values into a date object.
```

```
 * If, on the basis of the parameters, the method cannot generate a valid date,
 * it returns "null".
 *
 * Creation date: (28 Feb 2001)
 * @author: John Johnson
 */
```

### 4.3 Comments on lines in a program – "Why does it do this?"

Within a method, you will often want to comment on individual lines of code. Here, it is important to describe "why" and not "what" (the "what" is obvious; the reader will see it for him- or herself):

```
//-- Good: Describes "why".
//-- if month counter runs over, correct it
while (month > 12) {
    month -= 12;
    year++;
}

//-- Bad: Describes "what".
//-- check whether "month" is greater than 12
while (month > 12) {
    month -= 12;
    year++;
}
```

### 4.4 Warning about dangers

Some program lines are especially sensitive because they harbor dangerous or bad-looking elements. Always add comments to warn about such elements:

```
if (isBasicVNDataFilled() && (!(isQuotation() && isVPIdentical()))) {
  //-- Warning HACK: Now we have either an application
  //-- or a quotation in which the VN != is the VP
  //-- (Sorry ... - the Papyrus side requires this ...)
  getPapyrusObjectStream().addPapyrusObject(getPolicyHoldersHomeAddress());

  ...
}

/**
 * This is automatically called and returns the (seven-character)
 * application number. -- Warning: Each call to this
 * method automatically increments the application number.
 *
 * Creation date: (14 Feb 2001), John Johnson
 */
protected String needNextNumApplication() {
    return QuotationProperties.getNextNumApplicationLifeFund();
}
```

### 4.5 Removing automatically-generated and worthless comments

VisualAge automatically generates comments that say nothing and are therefore worthless. You should eliminate them (they only irritate the reader). No comment is better than a pseudo comment:

```
/**
 * Bad: Automatically generated "empty comment shells"
 *
 * Insert the method's description here.  // A: Says nothing and is therefore
 *                                        // worthless!
 *
 * Creation date: (28 Mar 2001), John Johnson
 *
 * @return int                 // B: Is obvious and therefore worthless!
 * @param vmId long            // B: Is obvious and therefore worthless!
```

```
 * @param errorList int[]     // B: Is obvious and therefore worthless!
 * @param maxErrors int       // B: Is obvious and therefore worthless!
 */
public native int getErrorList(long vmId, int[] errorList, int maxErrors);
```

VisualAge automatically generates the above empty comment shells A and B. You should either delete them or (better) fill them with content:

```
/**
 * Good: A comment that contains something
 *
 * In "errorList", returns a list with error numbers for the virtual
 * machine "vmId"; the data buffer can hold up to "maxErrors" errors.
 *
 * Creation date: (28 Mar 2001), John Johnson
 *
 * @return int - Number of errors that in fact occured (can exceed
 *     "maxErrors").
 * @param vmId long -- Id of the virtual machine.
 * @param errorList int[] -- error-code buffer.
 * @param maxErrors int -- Maximum number of errors that the buffer "errorList"
 *     can hold.
 */
public native int getErrorList(long vmId, int[] errorList, int maxErrors);
```

### 4.6 Parts of a comment for a class

A comment for a class should *at least* contain the following parts (the same of course also holds for interfaces):

- Description of the class's purpose
- Creation date
- Author of the class (Javadoc key word *@author*)

```
/**
 * Class that facilitates the use of dates (instances of "Date").
 *
 * Creation date: (20 Jun 2000)
 * @author: John Johnson
 */
public class DateHelper {...}
```

If you need to, you can also use the following Javadoc key words, among others:

- Description of all changes (including date and author; see also the next section).

```
/**
 * Creation date: (18 Apr 2001)
 * @author John Johnson
 *
 * -- History ------------------------------------------------------------
 *
 * -- 09 May 2001, John Johnson
 *    Implementation of the interface "FormConstants" and use of the constants it
 *    contains for the HTML fields.
 *
 * -- 04 May 2001, Hanna Hansen, John Johnson
 *    Various adaptations due to the changeovers in the class IQSState.
 */
public class LifePensionQuotation extends IQSState implements FormConstants {
    public static final String STATENAME = "LifePensionQuotation";
    ...
}
```

- @see – Cross reference to another class
- @deprecated – Indication that the class or interface should no longer be used

### 4.7 Parts of a comment for a method

A comment for a method should contain *at least* the following parts:

- Description of the purpose of the method, including the meaning the parameters and of the return values in text form, in other words without using the corresponding Javadoc key words.
- If you add a method to a "foreign" class (one you didn't write yourself): Creation date and name of the (new) author (if this comment line is missing, it is to be assumed that the original creator of the class also wrote this method)
- Description of each change, including date and author). If you don't describe the change in the method itself, do so in the change list for the class (see the previous section).

```
/**
 * Returns "true" if "year" is a leap year (otherwise "false").
 *
 * Creation date: (10 May 2000), Joe Zawinul
 *
 * -- History -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
 *
 * -- 23 Jun 2000, Jolanda Lauda
 *    Unnecessary variable "fOk" removed.
 *
 * -- 10 Jun 2000, Bruno Braun
 *    Error corrected (every year evenly divisible by 400 is a leap year).
 */
public boolean isLeapYear (int year) {...}
```

If you need to, you can also use the following elements and Javadoc key words, among others:

- Creation date and author of the method (see the comment in the list above)
- @param – one for each parameter
- @return – if the return value is not void
- @exception – one for each exception
- @see – cross reference to another method
- @deprecated – if the method should no longer be used

### 4.8 Commented-out code

You should generally not leave "commented-out" code in the source code, because it significantly hurts the active code's legibility. Each save operation in VisualAge generates a separate edition. This makes earlier code accessible at any time:

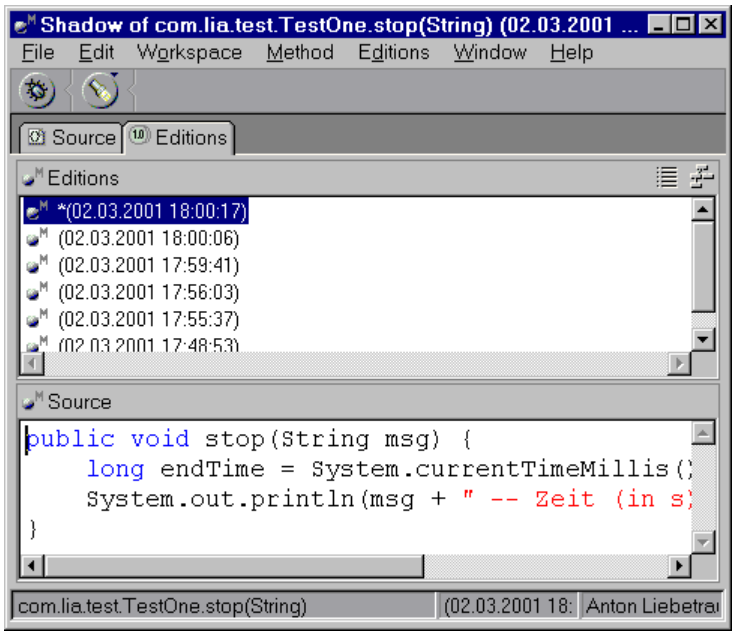


**Figure 8:** The successive editions of the method *stop(...)*.

### 4.9 Classes and methods without comments spread little joy

A class or method without meaningful comments is incomplete and must not in any case be versioned or released: missing comments lead to unnecessary costs, and hamper maintenance and further development. When documenting, please observe the following points:

- Versions with a leading zero (such as 0.33 or 0.4.1) can be versioned and released *in emergencies* even if they lack comments, since the leading zero shows that these versions are not yet finished (or released). It is nevertheless a good idea to version even these versions, while they are still fresh in your mind (see also the comments on versioning in Chapter 6, Versioning.)
- Some noted experts even recommend documenting all classes and methods before implementing the code. This has the desirable side-effect of ensuring that even before starting to code, the developer knows exactly the requirements he or she must fulfill (Assumption: not all problems were solved in detail in the preceding analysis and design phase.)
- Anyone who versions undocumented classes, interfaces or methods will be declared Devil of the Day and locked in the bathroom for two hours.

**See also:** Sections 32-66 in [RogueWave2000].

# 5. Access methods for attributes

Always declare class and instance variables as *private* to prevent direct access from outside. (For well-known reasons, only methods should access such variables.) In constrast, you can declare a constant (*static final*) as *public* if it is also to be available outside the class (an unwanted side effect is impossible here).– You can very easily have VisualAge generate the needed access methods.

As is well known, each attribute can have two access methods (but these are not required); these access methods usually bear the same name as the attribute (extended with the prefix *set* or *get*). For promote readability, do not place attribute prefixes in the names of the access methods:

```
private static Boolean mstBigEndian = null;          // Attribute with prefix
                                                     // "mst"

public static void setBigEndian(Boolean val) {...}  // Setter method without
                                                     // "mst"
public static Boolean getBigEndian() {...}           // Getter method without
                                                     // "mst"
```

## 5.1 Generation of access methods

To have VisualAge generate the access methods, select the desired class and click on the menu item *Generate, Accessors...* (It doesn't matter which browser displays the class.) The following dialog now appears:

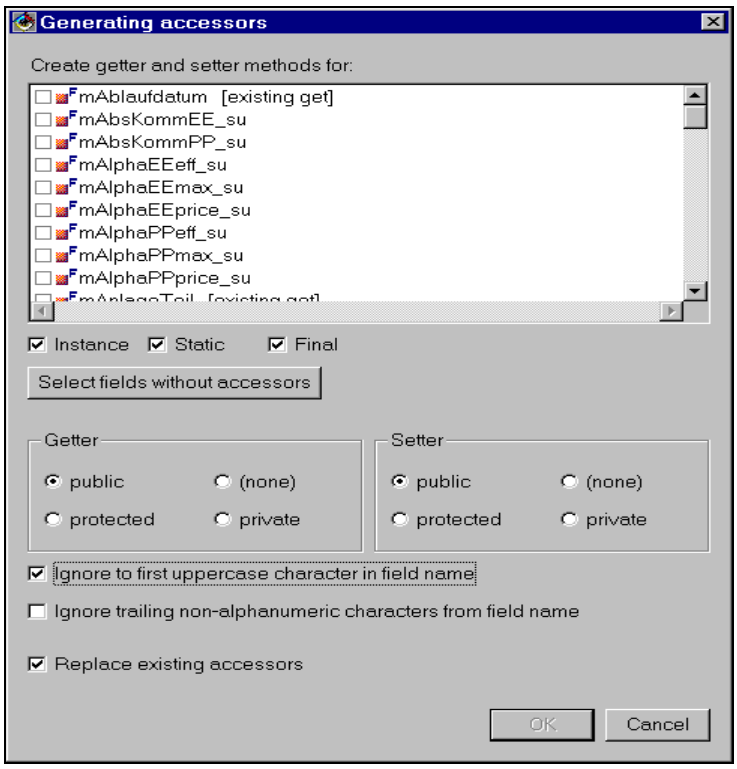


**Figure 9:** In this dialog, you can influence the generation of the access methods. With the option *Ignore to first uppercase ...*, for example, you can prevent attribute prefixes from appearing in the names of the access methods you generate.

In the list *Create getter and setter methods...* select all attributes for which you want to generate access methods.

In the sections *Getter* and *Setter*, you can specify the method modifiers. (For example, in the method *public Date getAblaufdatum()*, *public* is the method modifier). The option *(none)* creates an access method *without* a modifier (but it *does* create the access method!). VisualAge thus generates the two access methods, but you possibly must later erase them by hand.

If you activate the checkbox *Ignore to first uppercase ...* , VisualAge ignores leading lowercase letters and special characters (such as "_") in an attribute's name when generating the access methods. For example, if an attribute's name is *mName*, then the name of the access methods will be *getName* and *setName.* (If you don't activate the option, the names will be *getMName* und *setMName.*) – This option is ideal for eliminating our prefixes "mst" and "m".

If you activate the checkbox *Ignore trailing ...* , VisualAge ignores special characters at the end of the attribute name when generating the access methods. For example, if an attribute's name is *name_*, then the name of the access methods will be *getName* and *setName.* (If you don't activate the option, the names will be *getName_* and *setName_.*) – I currently see no way to use this option.

### 5.2 JavaBean convention

According to the JavaBean convention, the attribute access methods must start with *set*, *get* (read) or *is* (read for boolean attributes). Some examples:

```
setName(String val)        // Sets the string attribute "mName"
getName()                  // Reads the string attribute "mName"
setBirthDate(Date val)     // Sets the date attribute "mBirthDate"
getBirthDate()             // Reads the date attribute "mBirthDate"
setParent(boolean val)     // Sets the boolean attribute "mParent"
isParent()                 // Reads the boolean attribut "mParent"
```

The JavaBean convention also (unwisely) prescribes, that the getter method of a boolean attribute *always* must start with *is* (this is not particularly practical; see also section 3.3, Methods, in which we noted that the names of boolean getter methods usually start with the prefixes *is*, *has* or *can*). To comply with the JavaBean convention, be careful when naming boolean attributes.

#### 5.2.1 Bad name choices

In the following class *FirstTry*, the names of the boolean attributes are not particularly well-chosen (even though they are self-explanatory):

```
public class FirstTry {
  private boolean mIsParent;
  private boolean mHasChildren;
  private boolean mCanClose;
}
```

According to the JavaBean convention, the access methods for these attributes should have the following names (which is of course unacceptable; and when generating access methods, VisualAge follows the JavaBean convention):

```
setIsParent(...)
isIsParent()
setHasChildren(...)
isHasChildren()
setCanClose(...)
isCanClose()
```

#### 5.2.2 Good name choices

To obtain reasonable access method names, you can give the boolean attributes names like the following:

```
public class SecondTry {
  private boolean mParent;
  private boolean mHavingChildren;
  private boolean mAbleToClose;
}
```

The attributes of the class *SecondTry* lead to the following access-method names (which are meaningful, and what VisualAge automatically generates):

```
setParent(...);
isParent();
setHavingChildren(...);
isHavingChildren();
setAbleToClose(...);
isAbleToClose();
```

**Comment:** In the past, we haven't worried about the Java-Bean konvention. We suggest that you do so only if necessary.

**See also:** Section 24 in [RogueWave2000].

# 6. Versioning

As is well known, within the VisualAge development environment, you can assign versions to projects, packages, classes and interfaces. You can also "release" packages, classes, and interfaces:

- **Versioning:** Procedure that assigns a unique name to a project, package, class or interface; the name consists of a number or a text. Each versioning creates an independent version that can be restored at any time.
- **Release:** Declaration of some version of a package, class or interface to be the current version. – Projects cannot be released.

While you are working, projects, packages, classes and interfaces are in an open (temporary) state (*open edition*). The versioning process freezes this state; you must assign a number or a text to the resulting version:

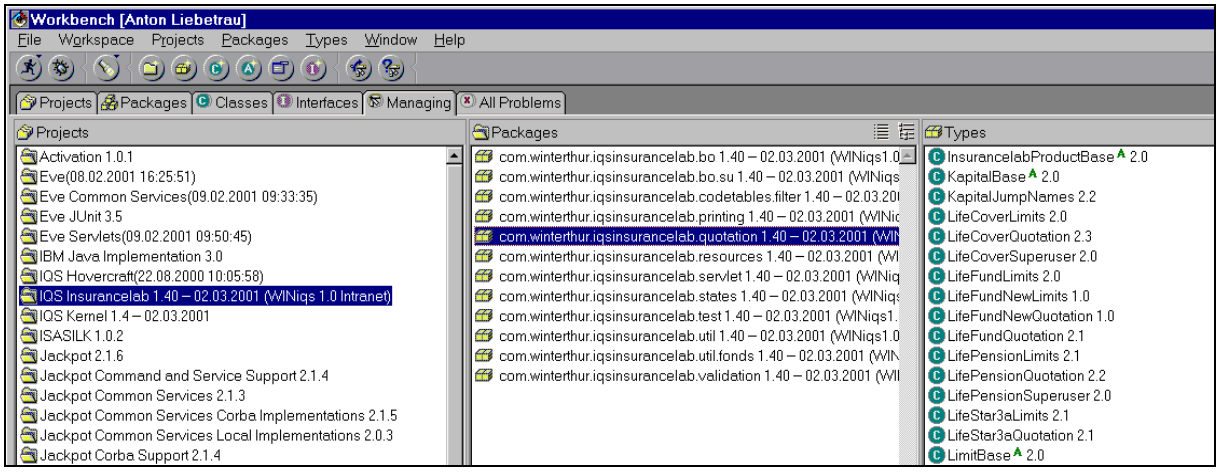


**Figure 10:** The "Managing" tab of the VisualAge workbench displays projects, packages and classes with their version numbers (and version texts). For example, the project "Eve" is in an open state (as the date and time to the right of the project name indicate), whereas the project "Jackpot" is versioned (Version 2.1.6).

Die version numbers of projects, packages, classes and interfaces do *not* necessarily have be synchronized (identical) with each other. In the figure above, the package "...quotation" has version 1.40 (and an additional text), whereas the classes it contains are all in the version 2.x. – Naturally, an application's manager has the option of synchronizing the version numbers of all of an application's projects, packages, classes and interfaces before the application's delivery.

### 6.1 Versioning of projects

Shortly before the delivery of an operational version of a project, the project is typically given a name (consisting of a number or a text); this is called versioning. You cannot version a project unless all the packages, classes and interfaces it contains are versioned and released.

In the tab "Projects" of the VisualAge workbench, you can add a comment to a project, but you must do so *before* you version the project. In addition to a general description of the contents of the project (*project content*) you can also mention dependencies on other projects (*prerequisites*) and changes (*change history*):

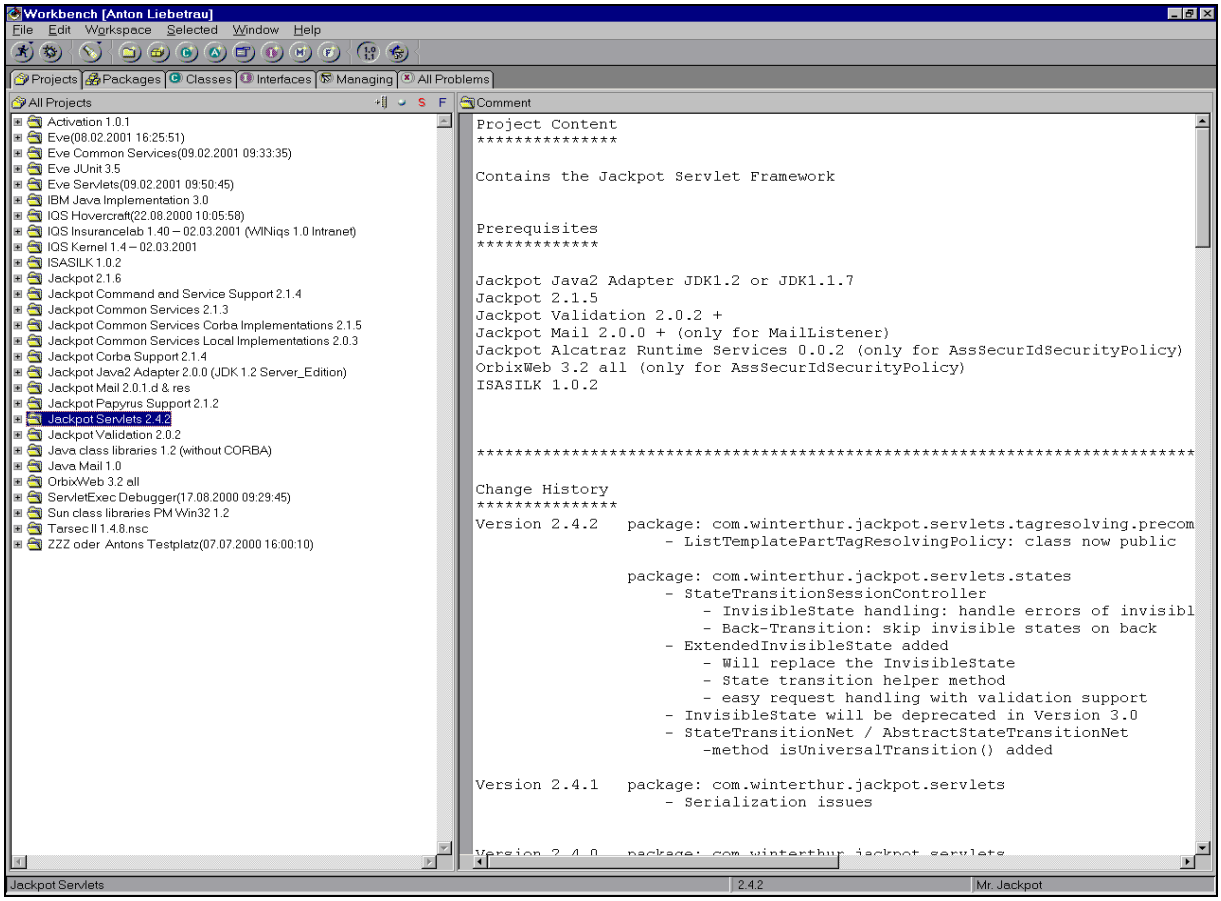


**Figure 11:** In the tab "Projects" of the VisualAge workbench, you can add comments to a particular project.

**Comment:** Up through version 3.0 of VisualAge, the best place to state dependencies on other projects (*prerequisites*) was in the project comments. From version 3.5 onward, however, the best place for such dependencies is the *Solutions* section.

### 6.2 Versioning of packages

Shortly before the delivery of an operational version of a package, the package is – like a project – typically given a name (consisting of a number or a text) and then released. You cannot version a package unless all the classes it contains are versioned and released.

### 6.3 Versioning of classes and interfaces

Classes and interfaces are versioned and released on a ongoing basis (in other words, not just before the delivery of an operational version, but *each time* the class or interface reaches a useful state). In practice, different numbering systems are used when versioning classes and interfaces:

- Format option "X.Y [Text]" for applications
- Format option "U.V.W [Text]" for frameworks
- Format option "A.B.C.D [Text]" for software firms
- Format option "Z [Text]" for minimalists

The following sections briefly describe these format options.

### 6.3.1 Format option "X.Y [Text]" for applications

The version number of type "X.Y [Text]" consists of two whole numbers (and an optional text). This format option has proved useful in *application development*, because it is both easy to use and sufficiently meaningful.

- **X** – Is normally increased by at least 1 after conceptual changes or major reworking (a 0 indicates that the class or interface is still under development).
- **Y** – Is usually increased by at least one after changes such as the addition of a new method, change in a parameter list, or correction of errors (corresponds to the "V" and "W" in the following format option for versioning).
- **Text:** Optional; any desired comment on the version.

**Comment:** We recommend that you not advance the version number "X" from 0 to 1 until you have made the class or interface ready for general use (in other words, until you have made all necessary tests and written all comments).

### 6.3.2 Format option "U.V.W [Text]" for frameworks

The version number of type "U.V.W [Text]" consists of three whole numbers (and an optional text). This format option has proved useful for *framework development.*

- **U** – Is normally increased by at least 1 after conceptual changes or major reworking (a 0 indicates that the class or interface is still under development).
- **V** – Is increased by at least 1 after changes like additions of new methods or changes in parameter lists.
- **W** – Is increased by at least 1 after correction of errors (provided the errors involved no significant changes such as new methods or changes to parameter lists).
- **Text:** Optional; any desired comment on the version.

**Comment:** Here as well, we recommend that you not advance the version number "U" from 0 to 1 until you have made the class or interface ready for general use (in other words, until you have made all necessary tests and written all comments).

### 6.3.3 Format option "A.B.C.D [Text]" for software firms

This versioning type is often used by large software houses; its force of expression is too detailed, however, to be appropriate for use here at Winterthur.

### 6.3.4 Format option "Z [Text]" for minimalists

The force of expression of this versioning type is rather modest. In practice, you should not use it for classes and interfaces.

### 6.3.5 Versioning in practice

In the course of our practical work with versioning of classes and interfaces, the following points have become clear:

- **Format option for versioning:** For the I-Solutions team (VICS 3), we recommend for versioning the uncomplicated and appropriate format option X.Y [Text].
- **Invalid version:** Append a "sh" (for "shit happened") to an invalid or unsuccessful version.
- **Consistency:** A consistent format option for versioning should be used for classes and interfaces throughout a project.

- **Version number 0:** A main version number of 0 (such as *0.77*) indicates that the class or interface is still in the development phase and thus has not yet been released.

# 7. Prefixes and suffixes

In assigning names to classes, methods and variables, use prefixes and suffixes whenever possible to improve the legibility of program code.

## 7.1 Prefixes and suffixes for classes

For many class names, we use prefixes (such as *Bo* and *Res*) or suffixes (such as *Quotation* and *Controller*). Prefixes normally consist of standardized abbreviations; suffixes, in contrast, tend more to consist of nouns written out in full. Whether you use prefixes or suffixes is mostly a question of taste – both are used heavily in practice.

Prefixes usually are more meaningful and provide a better overview than suffixes, but also tend to be more cryptic (the software developer must first learn them).

### 7.1.1 General prefixes

Here is a list of general prefixes.

| Prefix | An example | Comment |
|---|---|---|
| Bo | BoContract | Bo = business object |
| BoMap | BoMapUser | Map = mapping |
| BoSu | BoSuProduct | Su = super user |
| Ct | CtCountries | Ct = code table |
| Ctrl | CtrlLifeStar3a | Ctrl = controller |
| Doc | DocCustomerApplication | Doc = document |
| Res | ResErrGui | Res = resources |
| So | SoCustomer | So = server object |
| St | StLifeCoverQuotation | St = state |
| Val | ValLifeStar3a | Val = validator |
| Vw | VwContract | Vw = view |
| Wo | WoQuotation | Wo = work object |

As the above prefixes *BoMap* and *BoSu* show, you can combine several prefixes.

### 7.1.2 Project-specific prefixes

A project often establishes its own class prefixes:

| Prefix | An example | Comment |
|---|---|---|
| Fd | FdCapitalProduct | Fd = form descriptor |
| Lc | LcPlanOverview | Lc = large company |
| Sc | ScCompanyOverview | Sc = small company |
| Lf2 | Lf2GeneralInputs | Lf2 = product "New Life Fund" |

Each project can (and should) define its own class prefixes as needed.

## 7.2 Prefixes for methods

As Chapter 5 explains in more detail, an access method for an attribute can always use the prefix *set*, *get*, *is*, *has* or *can*. In addition to the prefixes for access methods, here are some of the other prefixes you will often encounter:

| Prefix | Possible meaning |
|---|---|
| test | test method (see also [EveTest2001]) |
| register | registration of a listener or a service |
| add | addition of an element to a list (position irrelevant) |
| append | addition of an element at the end of a list |
| insert | insertion of an element within a list (at a particular position) |
| delete | deletion of an element (at a particular position) |
| read | read out of a file or a stream |
| write | write to a file or a stream |
| open | open a file or a stream |
| close | close a file or a stream |

Personally, I often use the prefix *need* to indicate that a method is automatically called and must return some value (this is a so-called template method; see also [Gamma1995]). An example:

```
/**
 * Is automatically called whenever the product name is needed.
 * This name must be returned as a function value.
 *
 * Creation date: (1 Feb 2001), John Johnson
*/
protected abstract String needProductName();

/**
 * Is automatically called when a "jump target" for
 * error correction is needed (should a limit be exceeded).
 *
 * Creation date: (6 Mar 2001), John Johnson
 */
protected abstract ListResourceBundle needJumpNameInstance();

/**
 * Is called automatically and must return the standard form descriptor
 * (or null, if none exists).
 *
 * Creation date: (22 Nov 2000), John Johnson
 */
public abstract HtmlFormDescriptor needDefaultHtmlFormDescriptor();
```

### 7.3 Prefixes for variables

With the term "variable", we mean a class attribute, a method parameter, or a local variable. In addition to the mandatory "mst" and "m" for class attributes (see section 3.4, Attributes) still other (optional) prefixes for variables are certainly imaginable, namely ones that provide an indication of the variable's data type – this is where the so-called "Hungarian notation"[5] comes in; it should be familiar to inveterate C- and C++ programmers. The following table presents frequently-used prefixes.

| Data type | Prefix | An example | Comment |
|---|---|---|---|
| boolean | f | fOk | f stands for "Flag" |

[5] The «Hungarian notation» was developed in the early 1970s by Charles Somonyi and got its name mostly from the notation's cryptic-seeming prefixes (and of course also because Charles Somonyi was originally Hungarian). – See also [Maguire1993].

| Data type | Prefix | An example | Comment |
|---|---|---|---|
| char | c | cKey | |
| byte | b | bReadVal | |
| short | sh | shCountryCode | Only needed in special cases (normally, you should use "int" or "long"). |
| int | i | iAmount | |
| long | l | lPersId | |
| float | – | – | We recommend that instead of "float" you always use "double", given that the precision of calculations with "float" values is not fully satisfactory. |
| double | d | dInterestRate<br>dPremium | |
| String | s | sLastName | |
| StringBuffer | sb | sbHtmlOutput | |
| Date | dt | dtStart | |
| Hash table | ht | htJumpTable | |
| Vector | vc | vcFund | |
| [] (= Array) | ar | byte[] arbBuffer<br>int[] ariShares<br>double[] ardInterest | The prefix for an array variable consists of "ar" and the prefix of the array elements. |

**Comment:** These variable prefixes are optional and – as we already mentioned – you can use them according to your own tastes. If you use them together with the attribute prefixes, write the attribute prefixes first:

```
private static boolean mstfBigEndian = true;
private int miCount;
private byte[] marbBuffer = new byte[1024];
```

# 8. Tips for Programming

This chapter presents various tips and tricks for programming. They mostly came to light during our reviews of the applications "wincolink planet" and "iqs 4 insurancelab".

## 8.1 Naming of elements

Give self-explanatory names to classes, interfaces, methods and variables. The name of a class, method or variable (and not just the documentation) should make clear the element's (main) purpose (see also sections 18–30 in [RogueWave2000]).

### 8.1.1 Names for classes

A class name is usually a noun. Here are a few good examples (see also section 7.1, Prefixes and suffixes for classes):

- *DataValidator* – A class that mainly validates data (suffix notation).
- *DataHolder* – A class that mainly stores data (the class may also validate data, but this is not its main purpose; suffix notation).
- *BoContract* – A business-object class for storage of contract data (prefix notation).
- *CtCountries* – A code-table class for countries (prefix notation).

### 8.1.2 Names for interfaces

An interface name usually consists of nouns or adjectives. Some good examples:

- *Calculator, UserProfile* – Nouns; describe certain objects or things.
- *ValidationCases, JackpotConstants* – Nouns; collection baskets for constants (static, final attributes)
- *ActionListener, ExceptionHandler* – Nouns with the ending "er"; describe services.
- *Clonable, Runnable, Accessible* – Adjectives with the ending "able" or "ible"; describe capabilities.

### 8.1.3 Names for methods

Ideally, a method name consists of a verb and noun (constructors are of course an exception here, given that they must bear the same name as the class itself). Here are some good and problematic examples (see also section 7.2, Prefixes for methods):

- *needProductName, setName, addListener, doScreen* – Verb and noun (good names)
- *focusLost, languageChanged, keyPressed* – Noun and verb (good names)
- *perform, finalize, initialize, consume* – Verb (good names)
- *error, screen, doorbell* – Nouns (problematic: meaning of the method is unclear, much better would be: *writeError, refreshScreen, ringDoorbell*).

### 8.1.4 Names for variables

Variable names should be as short as possible (and thus easy to handle) but also meaningful: Here are some good and problematic examples (see also section 7.3, Prefixes for variables):

- *mName, mMaxEntryAgeWop, mstInstCount* – Meaningful names for class attributes with the prefix "m" or "mst" (good names).
- *dTax, sbHtmlLine* – Meaningful variable names with prefixes (good names)

- *tax, htmlLine* – Meaningful variable names without prefixes (good names)
- *i, j, k, val, flag, res, fos, dis* – short variable names that say little (problematic names; they are okay as loop variables and within very short blocks or methods, but otherwise bad – by short blocks or methods I mean up to five lines.)

### 8.2 Control variables in "for"-loops

In a *for*-loop, you can change the value of a control variable in the loop's header, but never within the loop, for the following reasons:

- You circumvent the termination condition in the loop header.
- You make the code unclear and prone to errors.

The following (rather implausible) example clarifies this problem:

```
//-- Very bad code:
String sName = "Hoho__Hihi__Huhu__";
StringBuffer sbName = new StringBuffer();

for (int i=0; i<sName.length(); i++) {
  try {
    while (sName.charAt(i) == "_") i++;  // Not like this! Here comes the noise!
    sbName.append(sName.charAt(i));
  } catch (StringIndexOutOfBoundsException e) {
    //-- The thing blew up!
  }
}


//-- This code is okay.
String sName = "Hoho__Hihi__Huhu__";
StringBuffer sbName = new StringBuffer();

for (int i=0; i<sName.length(); i++) {
  if (sName.charAt(i) == "_") continue;  // Go immediately to the next iteration
  sbName.append(sName.charAt(i));
}
```

### 8.3 Using the word "this"

The reserved word "this" is useful in two situations: you can use it in a constructor, to call another constructor of the same class, or place it before an instance attribute or instance method. An example:

```
public class TheUseOfThis {
  private String lastName;
  private String firstName;
  private String mLastName;

  public TheUseOfThis() {
    //-- Call to another constructor of the class "TheUseOfThis"
    this(null);                  // A: good (no other way to do it)
  }

  public TheUseOfThis(Object obj) {
    super();
    ...
  }

  public void setNameEtCetera(String lastName) {
    //-- Assign the parameter "lastName" to the attribute "lastName" or
    //   "mLastName"
    this.lastName = lastName;     // B: good (no other way to do it because of
                                  //   the name conflict)
    this.firstName = "John";      // B: good ("this" == access to attribute)
```

```
      firstName = "John";           // B: avoid

      this.mLastName = LastName;    // C: avoid (unnecessary extra weight)
      mLastName = lastName;         // C: good (prefix "m" == access to attribute)

      //-- Call the class's own method
      this.setParent(false);        // D: avoid (unnecessary extra weight)
      setParent(false);             // D: good
    }

    ...

  }
```

We recommend the use of "this" as follows:

| Placement of "this" | When to use | See above example |
|---|---|---|
| In a call to a constructor within a constructor | Whenever needed. | A |
| At the start of an attribute name (if it lacks a prefix) | Whenever you want to show that the attribute (and not a local variable) is accessed. | B |
| At the start of a attribute name that already has the prefix "m" or "mst": | Never (or only in case of name conflicts – which the prefixes themselves should prevent from occurring). | C |
| At the start of a method name | Never (it is unnecessary extra weight that delivers no new information). | D |

### 8.4 Concatenating strings

If you build a string from a number of sub-strings (in a loop, for example), use the class "StringBuffer" instead of the class "String" for reduced memory use and better performance:

```
//-- A: Slow and memory-hungry version
public String getUselessString(int iCount) {
  String sTest = "use";
  for (int i=0; i<iCount; i++) {
    sTest += "-less";  // slow, memory-hungry
  }
  return sTest;
}

//-- B: Better version:
public String getUselessString(int iCount) {
  StringBuffer sbTest = new StringBuffer("use");
  for (int i=0; i<iCount; i++) {
    sbTest.append("-less");
  }
  return sbTest.toString();
}
```

In example A above, each loop iteration generates a string "corpse" (since, according to the Java definition, strings are constant and cannot be changed); the garbage collector must later remove it. In example B, this doesn't occur. – The following table compares the run-time behavior of the variants A and B. Whereas variant A quickly brings the system to its knees (apparently by memory starvation), variant B does the job fast – even if many iterations are involved:

| Value of "iCount" | Variant A (Duration, in seconds) | Variant B | B fast than A (Factor) |
|---|---|---|---|
| 1,000 | 0.090 | (to fast) | (?) |
| 5,000 | 1.530 | 0.010 | 153 |
| 10,000 | 6.570 | 0.030 | 219 |
| 50,000 | (not measurable) | 0.070 | (?) |
| 100,000 | (not measurable) | 0.190 | (?) |

### 8.5 Constructors

As is well known, a constructor generates instances from a class, and initializes the instances. A class can have any number of constructors. Each constructor must, however, have a different parameter list:

```
public Randomizer() {                 // Default constructor
  this(System.currentTimeMillis());   // Call to another constructor
}

public Randomizer(long seed) {        // Constructor that does the work
  super();
  mRandseed = seed;
}
```

In the case of constructors, please note the following:

- *Default constructor:* By default constructor, we mean the constructor with no parameter list. If a class has no constructor at all, Java automatically adds a default, public constructor for it.
- *Preventing the creation of an instance:* To prevent the creation of instances for a class (as for the class *Math*), set the default constructor to *private*. – Warning: Classes without constructors automatically have a default constructor (which of course allows generation of instances; see the previous point).
- *Several constructors:* If a class has several constructors, one single constructor should take care of the main action (for example initializing instances). To avoid duplication of programming work, the rest of the constructors should call the main constructor directly or indirectly – that is, by way of other constructors (see the example above).

### 8.6 Constants ("static final")

In Java, a constant is a *static* and *final* attribute (see section 3.4, Attributes). You can define constants in classes or interfaces:

```
public class CtFinancingStyle extends ListBundleBase {  // Class with constants
  //-- Constants of the code table
  public static final int UNDEF = 0;
  public static final int EE    = 1;
  public static final int PP    = 2;
  public static final int MIXED = 3;

  //-- Contents of the code table (default language)
  static final Object[][] contents = {
    {UNDEF+"", "-- Please select --"},
    {EE+"", "Lump-sum deposit"},
    {PP+"", "Premiums"},
    {MIXED+"", "Lump-sum deposit and premiums"}
  };
}
```

```
public interface FondsFormConstants {  // Interface with constants
  //-- Background color for the selected color
  public final static String COLOR_DE             = "ColorDE";
  public final static String COLOR_FR             = "ColorFR";
  public final static String COLOR_IT             = "ColorIT";
  public final static String COLOR_EN             = "ColorEN";

  //-- General constants
  public final static String RESET                = "reset";
  public final static String BASE_ICC_URL         = "BaseICCURL";

  ...
}
```

### 8.6.1 Constants in classes

If you define the constants within a class, you can access them as follows:

```
import com.winterthur.eve.codetables.CtFinancingStyle;

public class Test {
  private int mFinancingStyle = CtFinancingStyle.EE;  // Place class name before
                                                      // constant name
  ...
}
```

### 8.6.2 Constants in interfaces

Here's how to access a constant within an interface:

```
import com.winterthur.eve.const.FondsFormConstants;

public class Test implements FondsFormConstants {     // Implementation of
                                                      // interface
  private String mDefaultColor = COLOR_DE;
  ...
}
```

### 8.6.3 Constants in classes or interfaces

Whether you define constants in classes or interfaces is mainly a matter of personal taste. But you should keep in mind a few points:

- To retain a overview, place each constant in the most closely-related class or interface; don't place a large number of unrelated constants in the same class or interface.
- Suppose two interfaces contain the same constant (for example, *public static final int UNDEF = 0;*). If you implement both these interfaces in the same class, you must place the interface name before the constant name. Examples:

```
public interface ConstOne {
  public final static int UNDEF = 0;
  public final static int EE = 1;
  public final static int PP = 2;
  public final static int MIXED = 2;
}

public interface ConstTwo {
  public final static int UNDEF = 0;
  public final static int MALE = 1;
  public final static int FEMALE = 2;
}

public class TestOne implements ConstOne, ConstTwo {
  private int mCount = ConstOne.UNDEF;   // Place interface name before constant
                                         // name
}
```

### 8.6.4 Use of constants

Use constants as often as possible. Do not place a constant character string (such as a key for an error message) or constant numeric value (such as the maximum allowed number of funds or the minimum entry age) in a code. Instead of a code, *always* use a constant.

### 8.6.5 Naming of constants

You should (of course) give constants self-explanatory names. With bad (and repeatedly-used) names like *ERROR_1* and *ERROR_2,* you make your life unnecessarily difficult. Much better here are names such as *LOADING_PRODUCT_ERROR* and *CALCULATION_ERROR*. (See also section 8.1, Naming of elements).

## 8.7 Comparisons

In a program, comparisons are pretty much everywhere. The following sections present a few of the major dangers in this area.

### 8.7.1 Data type of the comparison

The result of a comparison is well known to be of type *boolean*. You can either store this result in a variable, or return it directly as the value of a function:

```
if (i < MAXLEN) {     // avoid (unprofessional)
  return true;        //
} else {              //
  return false;       //
}                     //

return (i < MAXLEN);  // good (short and to the point)
```

### 8.7.2 "true" and "false" in comparisons

Do not place the reserved words *true* or *false* in a comparison (makes an unprofessional impression):

```
if (fOk == true) {...}             // avoid (unprofessional)
if (fOk != true) {...}             // avoid (unprofessional)
if (fOk == false) {...}            // avoid (unprofessional)
if ((fOk == false) == true) {...}  // avoid (overzealous)

if (fOk) {...}                     // good
if (!fOk) {...}                    // good
```

### 8.7.3 Boolean values with three states

Sometimes, in addition to *true* and *false*, you also need the third state "unknown" (or "not initialized"). You can achieve this very easily with the *Boolean* wrapper class (former Smalltalk fans will recognize this trick). The following example should make this clear:

```
public class ThreeStateBoolean {
  private Boolean mBigEndian = null;  // 3 states possible (null, true, false)

  ...

  public boolean isBigEndian() {
    if (mBigEndian == null) {                       // A: Query the state
      String sMode = readBigEndianModeFromFile();   // B: Read the state
      mBigEndian = new Boolean(sMode);              // C: String -> Boolean
    }
    return mBigEndian.booleanValue();               // D: Boolean -> boolean
  }
}
```

In this code, line A checks (with the help of the attribute *mBigEndian*, which can store a boolean instance) whether the Big Endian mode is already known. If not, then line B reads the Big Endian mode – a string value – out of a file. Given that this read step is relatively slow, you

should only carry it out once. Then line C generates a boolean instance and initializes it with the string value that B read. Line D returns the boolean value that line C saved in the boolean instance (and not the boolean instance itself, because you can't directly use it in normal comparisons).

### 8.7.4 Formatting of nested "if" statements

Nested "if" statements are comparable to "switch"statements. Format them as follows:

```
//-- good formatting (equivalent levels are aligned with each other)
if ("back".equals(sTransaction)) {
  ...
} else if ("calculate".equals(sTransaction)) {
  ...
} else if ("superuser".equals(sTransaction)) {
  ...
} else if ("expert".equals(sTransaction)) {
  ...
} else {
  ...
}

//-- bad formatting (equivalent levels are not aligned with each other)
if ("back".equals(sTransaction)) {
  ...
} else {
  if ("calculate".equals(sTransaction)) {
    ...
  } else {
    if ("superuser".equals(sTransaction)) {
      ...
    } else {
      if ("expert".equals(sTransaction)) {
        ...
      } else {
        ...
      }
```

### 8.7.5 Comparisons of strings

Always compare strings by using the method *equals* (and never with the symbols == or !=):

```
String str;

if (str == "stone") {...}        // A: bad (doesn't do the job)
if (str != "rock") {...}         // A: bad (doesn't do the job)

if (str.equals("stone")) {...}   // B: watch out (error if "str" == "null")
if (!str.equals("rock")) {...}   // B: watch out (error if "str" == "null")

if ("stone".equals(str)) {...}   // C: good and safe (never generates an error)
if (!"rock".equals(str)) {...}   // C: good and safe (never generates an error)
```

When comparing a string constant with a string variable, for safety always write the constant first (see example C above).

### 8.7.6 Comparison of doubles

Java stores double values internally in the IEEE 754 format, which means that the representation of some values can only be approximate (for example, this format forms 0.1 as the sum of 1/16, 1/32, 1/256, and so on, until it reaches an approximation of 0.1). For this reason, never compare double values with == or != :

```
if (dFac == 3.14159) {...}                    // dangerous
if (Math.abs(dFac - 3.14159) <= 0.001) {...}  // good (safe)
if (dVal >= 10000.0) {...}                    // good (safe)
```

### 8.7.7 Comparison of integers

In comparisons involving integer data types (*int, long ...*) little can go wrong. Comparisons that use the symbols == or != are entirely safe (in contrast to *double* values). Nevertheless, use them sparingly in termination conditions:

```
while (iPos != MAX_LEN) { ... }   // avoid (in case "iPos" never
                                  // becomes != MAX_LEN)
while (iPos < MAX_LEN) { ... }    // good
```

## 8.8 Converting strings into numeric values (and vice versa)

Converting strings into numeric values (and back into strings) is part of the daily bread of the harried software developer.

### 8.8.1 Converting strings into numeric values

You often need to convert strings into numeric values (such as *double, int or long*). Here are some ways to do so:

```
String sDouble = "1234.4";
double dVal1 = Double.valueOf(sDouble).doubleValue();  // long-winded
double dVal2 = Double.parseDouble(sDouble);            // good

String sInt = "1234";
int iVal1 = Integer.valueOf(sInt).intValue();          // long-winded
int iVal2 = Integer.parseInt(sInt);                    // good
```

You can program these conversions so that invalid characters in the strings trigger exceptions. Also, the format of the double value is country-dependant. Depending on the setting, the decimal separator is either a point or a comma. Here is code for a safe conversion:

```
try {
  dVal = Double.parseDouble(sInput);
} catch (NumberFormatException e) {
  dVal = 0.0;
}
```

### 8.8.2 Converting numeric values into strings

Converting numeric values into strings is simple and unassociated with any problems:

```
int iVal = 1234;
sWert = "" + iVal;                        // good
sWert = Integer.toString(iVal);           // requires some effort (but good)
sWert = Integer.toString(iVal, iBasis);   // good (if "iBasis" != 10)
sWert = Integer.toBinaryString(iVal);     // good (simpler you can't get)
sWert = Integer.toHexString(iVal);        // good (simpler you can't get)
sWert = Integer.toOctalString(iVal);      // good (simpler you can't get)

double dVal = 567.8;
sWert = "" + dVal;                        // good
sWert = Double.toString(dVal);            // requires some effort (but good)

int a = 1;
int b = 2;
sWert = "" + a + b;                       // returns the string "12"
sWert = "" + (a + b);                     // returns the string "3"
```

## 8.9 Import statements

Import statements can make your code easier to read. Without import statements, you must fully qualify the name of a class or interface by preceding it with the package name. Here is an example with *java.util.Date*:

```
//-- Avoid: hard to read
public class PersistentProperties extends java.util.Properties {
```

```
  private String mFileName;
  private java.io.FileInputStream mFis;
  private java.io.FileOutputStream mFos;
  private java.util.Date mLastActionDate;

  public java.util.Date getLastActionDate () {}
  ...
}

//-- Good: provides an overview
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.Properties;
import java.util.Date;

public class PersistentProperties extends Properties {
  private String mFileName;
  private FileInputStream mFis;
  private FileOutputStream mFos;
  private Date mLastActionDate;

  public Date getLastActionDate() {}
  ...
}
```

When using import statements, please keep the following in mind:

- The import statement does not really import classes or interfaces, but instead just makes them visible to the compiler. Two examples:

```
import java.util.Date;     // Makes the class Date within java.util visible
import java.util.*;        // Makes all classes and interfaces within java.util
                           // visible
```

- As is well known, you do not need to import the classes and interfaces of the package *java.lang* (they are always visible).
- When you're reading code, you want to know in what package a class or interface is located. Therefore, when you're writing code, never import whole packages. This makes for more writing, but helps the reader (and, sooner or later, all writers are readers):

```
import java.util.Date;                        // good
import java.rmi.server.UnicastRemoteObject;   // good
import java.util.*;                           // avoid whenever possible
import java.rmi.server.*;                     // avoid whenever possible
```

  Various experts have stated that it's okay to import complete standard Java pages such as *java.util* and *java.rmi.server*.

- You can import a whole package only when all its classes and interfaces bear the same, unambiguous prefix:

```
import com.winterthur.iqsinsurancelab.bo.*;  // good: contains only "Bo" classes
import com.winterthur.eve.codetables.*;      // good: contains only "Ct" classes
```

- When you save a method, VisualAge checks the source code and, in case of errors, automatically proposes corrections (see the following figure). One of the proposed corrections is to insert an import statement in the source code:

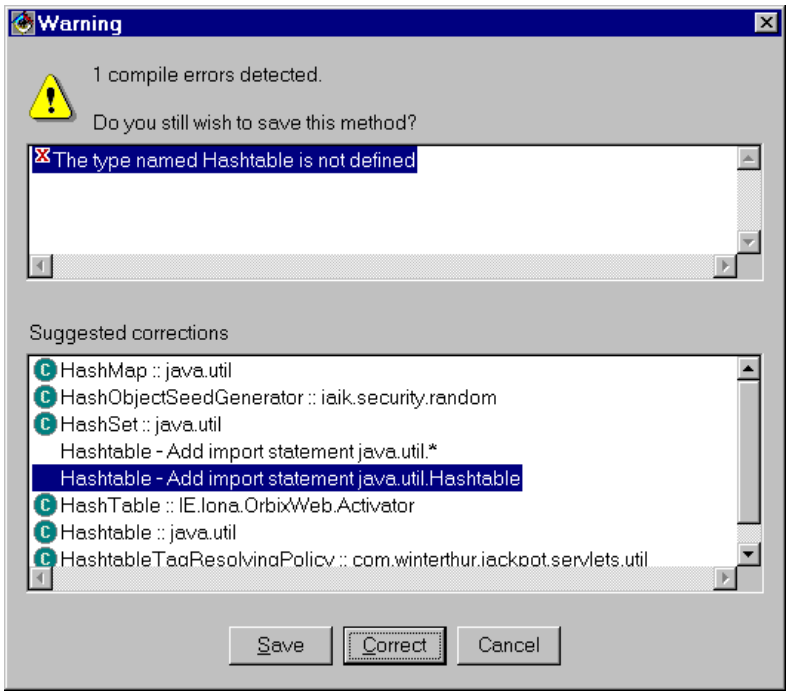


**Figure 12:** The *Correct* button carries out the proposed correction you select (here, it will insert an import statement in the source code).

# 9. Consolidating an application: tips

Applications are seldom error-free, much less perfect. For this reason, an application must be continually worked on (in other words cleaned up, disentangled, restructured, and freed of redundancies). This process is known as consolidation or "refactoring" and occurs during development of an application – immediately upon discovery of a weak point.[6] (Putting it off only makes the problem worse.)

Unfortunately, given the pressure of deadlines, it is not always possible to consolidate code enough before user tests or the delivery of an application. In this case, the development team must carry out the consolidation after the delivery deadline (assumption: consolidation is necessary). This phase must last at least a month, to leave enough time for serious reflection. After the consolidation phase, the user tests must be repeated. (You can automate some tests; see [EveTest2001]).

The following sections can help determine which parts of an application's code need further work (see also [Fowler1999]).

### 9.1 Code is hard to understand

- Problem: The code is hard to understand; even I, who wrote it, have some trouble understanding it.
- Solution: (Re-)document the methods and classes. In difficult cases, you may have to rework the code.

### 9.2 Code is hard to extend

- Problem: The code is hard to extend; I'm afraid to touch it.
- Solution: Restructure or rewrite the code.

### 9.3 Error-prone code

- Problem: Code errors always keep appearing in the same piece of code. To eliminate errors, new if-queries have been built into the code. This has led to a so-called "if mountain" handling a large number of special cases.
- Solution: Think out the code again and rewrite it.

### 9.4 Long lines (more than 80 characters)

- Problem: Lines contain more than 80 characters and thus hamper an overview. Often, the reason for this is that one line carries out complex inquiries, transformations or other operations. Two examples:

```
01: public void mySpecialMethodOne() {  // Example A-1
02:   ...
03:   //-- Bad: Rule for sale of 3a products in an "if" statement
04:   BoPerson pers = contract.getVn();
05:   if (pers.getNation() == CtCountry.CH || pers.getNation() == CtCountry.LI) {
06:   //-- 3a product can be sold
07:     ...
08:   }
09:   ...
10: }

01: public void mySpecialMethodTwo() {  // Example B-1
02:   ...
```

[6] [Beck2000], page 58: "You don't refactor on speculation [...]; you refactor when the system asks you to. When the system requires that you to duplicate code, it is asking for refactoring."

```
03:   //-- Bad: Data conversion in line of program code
04:   BoPayment paymnt = police.getPayment();
05:   double dPayment;
06:   try {
07:     dPayment = Double.parseDouble(paymnt.getDisabLevel());
08:   } catch (NumberFormatException e) {
09:     dPayment = 0.0;
10:   }
11:   ...
12: }
```

- Solution: Rework the code and place queries and conversions in methods. You should generally place such methods in the classes that store the data:[7]

```
01: public void mySpecialMethodOne() {  // Example A-2
02:   ...
03:   //-- Good: Rule for sale has now been placed in a method (details are
04:   //-- hidden)
05:   BoPerson pers = contract.getVn();
06:   if (pers.is3aProductAllowed()) {
07:     //-- 3a product can be sold
08:     ...
09:   }
10:   ...
11: }
```

```
01: public void mySpecialMethodTwo() {  // Example B-2
02:   ...
03:   //-- Good: Direct data conversion has been eliminated
04:   BoPayment paymnt = police.getPayment();
05:   double dPayment = paymnt.getDisabLevelAsDouble();
06:   ...
07: }
```

- Comment: The statements in examples A-1 and B-1 carry out various tasks, each of which involve some effort. It is certainly possible (and likely), that these statements occur several times in the code as a whole. This means that developers will forever be rewriting these evaluations and conversions. (This is, of course, undesirable and makes no sense.) – In general, a method should only have tasks that really belong inside it (as indicated by its name).
- Comment on example A: In example A-2, a new method, *is3aProductAllowed()*, extends the class *BoPerson*. This method has direct access to the person's nationality and hides (for the outside world) uninteresting business logic; it also keeps anyone from deciding to implement this business logic themselves in the future.
- Comment on example B: Example B-1 transforms a string into a double value. This tasks requires six lines (05–10); that's a lot for such a simple task. Also, the method *getDisabLevel()* returns – surprisingly – a string, even though the method's name implies it will return a double value. In the improved example B-2, the type conversion has been moved from the class *BoPayment* to the method *getDisabLevelAsDouble()*. (Even better, of course, would be if the method *getDisabLevel()* returned a double value right from the start.)

### 9.5 Long methods (more than 20 lines)

- Problem: A method is unwieldly, hard to see as a whole, or longer than 20 lines (not including comments).

[7] [Beck2000], page 48: "Good design puts the logic near the data [...]. Good design allows the extension of the system with changes in only one place."

- Solution: Split the method into several smaller methods.
- Comment: The 20 lines are a guideline only. – Longer methods often handle several different kinds of problems that do not necessarily belong together (such as initializing *and* validating data). Simply place each type of problem in its own (private) method. In exceptional cases, a method can of course be significantly longer than 20 lines, especially when the method carries out a series of similar tasks such as copying data or populating a data stream. Even in this case, however: Each method should deal with one kind of problem only.

### 9.6 Large classes (more than 20 methods)

- Problem: A class contains more than 20 methods (not including access methods for attributes).
- Solution: Split the class into several smaller classes.
- Comment: The 20 methods are a guideline only. – Often, design patterns can be helpful when restructuring code. It is thus essential to understand and use the most important patterns (see also below). – You already can make a number of improvements with the help of inheritance: If, within related classes (such as document, calculation or validation classes), the same or similar methods are always being implemented, an obvious solution is to implement corresponding general methods just once in an abstract superclass.
- A design pattern typically falls into one of three groups. Major design patterns are as follows:

  (1) **Creational pattern:** factory, singleton
  (2) **Structural patterns:** adapter, bridge, decorator, facade, proxy
  (3) **Behavioral patterns:** template method, strategy (= policy)

  See also the book [Gamma1995].

### 9.7 My eyes are burning (I work more than 42 hours a week)

- Problem: I fail to catch errors because I'm tired and overloaded.
- Solution: Do not accept to work overtime over longer periods without comment.
- Kent Beck writes the following on this subject: "Overtime is a symptom of a serious problem on the project. The [...] rule is simple – you can't work a second week of overtime. For one week, fine, crank and put in some extra hours. When you come in Monday and say, 'To reach our goals, we'll have to work late again today', then you already have a problem that can't be solved by working more hours."[8]

[8] [Beck2000], page 60.

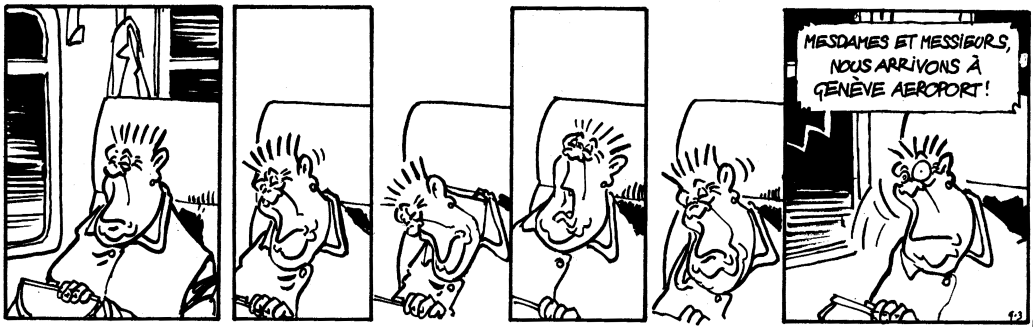


**Figure 13: "Ladies and gentlemen, we are arriving at Geneva Airport!"** Eve in train "IC 738"[9] on her way home from Winterthur to Zurich Airport, late in the evening after dozing off for a moment... (Jaermann/Schaad in the *Tages-Anzeiger* of 9 Mar 2001).

[9] As of May 2001, the «IC 738» ran daily from St. Gallen to Geneva Airport (17:10–21:40) and offered the following: Minibar or Railbar, and restaurant. Seat reservations were possible at any time (source: SBB Travel Online, www.sbb.ch).

## 10. For further reading

[AmbySoft2000] Scott W. Ambler: *Writing Robust Java Code*, 2000 (see http://www.ambysoft.com/javaCodingStandards.pdf).

[Beck2000] K. Beck: *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[e-Platform1999] e-Platform team: *Coding Guidelines*. Winterthur Insurance, 1999 (see http://c004020/ECPReference/jackpot/Codierungsrichtlinien/Codierungsrichtlinien.html).

[EveAppl2001] iSolution-Team: *Eve: The Application Framework.* Winterthur Life, 2001.

[EveTest2001] iSolution-Team: *Eve: Testing Java Applications.* Winterthur Life, 2001.

[Fowler1999] M. Fowler: *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[Gamma1995] E. Gamma et al: *Design Patterns.* Addison-Wesley, 1995.

[Maguire1993] S. Maguire: *Writing Solid Code.* Microsoft Press, 1993.

[RoqueWave2000] Rogue-Wave-Team: *The Elements of Java Style*. Cambridge University Press, 2000.

[Sun1999] Sun team: *Code Conventions for the Java Programming Language.* Sun Microsystems, 1999 (see http://java.sun.com/docs/codeconv/html/CodeConventions.doc.html).